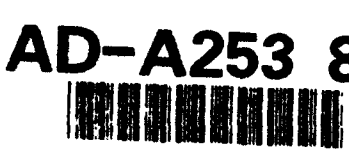


(1)

Deriving and Manipulating Module Interfaces

Robert Louis Nord

May 1992

CMU-CS-92-126

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of DARPA or the U.S. Government.

Keywords: formal methods, software development, software systems, integration, adaptation, program transformation, software components, module interfaces, abstract datatypes, data representations

School of Computer Science

DOCTORAL THESIS
in the field of
Computer Science

Deriving and Manipulating Module Interfaces

ROBERT LOUIS NORD

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

ACCEPTED:

MAJOR PROFESSOR	15 November 1991 **DATE**
MAJOR PROFESSOR	11/15/91 **DATE**
DEAN	5/1/92 **DATE**

APPROVED:

PROVOST	1 June 1992 **DATE**

Abstract

A formal method for systematically integrating general-purpose software modules into efficient systems is presented. The integration is accomplished through adjustment of abstract interfaces and transformation of the underlying data representations. The method provides the software designer with the ability to delay or revise design decisions in cases where it is difficult to reach an *a priori* agreement on interfaces and/or data representations.

To demonstrate the method, the development of a text buffer for a simple interactive text editor is given. For each basic operation on the text buffer, a natural and efficient choice of data representation is made. This organizes the operations into several "components," with each component containing those operations using the same data representation. The components are then combined using formal program-manipulation methods to obtain an efficient composite representation that supports all of the operations.

This approach provides meaningful support for later adaptation. Should a new editing operation be added at a later time, the initial components can be reused in another combining process, thereby obtaining a new composite representation that works for all of the operations including the new one. There are also ramifications for the application of formal methods to larger-scale systems, as this method can be applied to the manipulation of the interfaces between modules in larger software systems.

Contents

List of Figures

Acknowledgments

I thank the members of my committee, Peter Lee, William Scherlis, David Garlan, Nico Habermann, and David Notkin. Peter Lee and Bill Scherlis, my advisors, worked closely with me, teaching me how to do research, how to write and publish papers. The other members of my committee provided technical guidance. Their comments and suggestions have substantially improved the content and presentation of this dissertation. Many people have been instrumental in focusing the ideas in the thesis, I would especially like to thank Ira Baxter, Bernd Bruegge, Frank Pfenning, Gene Rollins, and Jeannette Wing.

This thesis grew out of my experience with the Ergo Project and has been influenced by my interactions with the members of the project. Thanks are due to Penny Anderson, Scott Dietzen, Conal Elliott, Tim Freeman, and Ulrik Jørring.

I want to thank my friend Howard, my mother and father, my grandparents, my brothers, Rick, Ran, Russ, and Eric, and all my relatives and friends for their encouragement, love, and support.

Chapter 1

Introduction

1.1 The Problem of Organizing Interfaces

The organization of interfaces among system components is a key task in the construction and management of larger-scale software systems. For many large systems, a principal source of risk is in making the decisions concerning the placement of these interfaces [11] — in other words, how the components are to be organized into an integrated software system. Language features for modularity, including various type systems, provide a means for component structure to be made more explicit, thus facilitating management of systems interfaces [65].

I suggest that formal methods [19, 31] can be applied to support the development and evolution of larger-scale systems through the formal manipulation of the interfaces and components. As a system architecture matures and evolves, interfaces and components will likely need to be adjusted in various ways, by moving or shifting computations across interfaces, by introducing new interfaces to create new components, by combining similar interfaces to merge components, and so on. Indeed, the architecture of large systems is rarely determined fully in advance and, in any case, evolves rapidly as development experience is gained. Within maintenance activities, for example, 60 percent of the effort is for enhancements [53]. Formal methods can provide a basis for the creation of software tools that support this kind of iterative refinement [4]. Such tools could potentially reduce the risks, which are now very high, associated with the determination of overall systems architecture in software development. The risks are high because of the need for enhancement

Consider the important problem of combining, or "integrating," modules that must share data. The possibility of sharing means that interacting modules must agree not only on the abstract interfaces, but also on the underlying data representations. Because architectures evolve, this problem of integration usually persists for as long as the system is maintained.

Consider, for example, the development of an interactive display-editor. A key subproblem is the implementation of operations on the text buffer. There are many possible representations for such buffers, for example a sequence of characters, a sequence of lines, and so on, and for each operation, one representation may be more natural or appropriate

than another. Rather than having to decide in advance on some compromise, it would be easier to collect into separate components the sets of individual editing operations that use, in a straightforward implementation, the same "natural" representations. I am proposing an approach that involves taking individual components, each using its own "natural representation," and combining them via program transformation into a single, efficient, composite implementation.

Performing this composition of components requires a way to mediate the interactions among them. Program-transformation techniques [25, 67] can provide assistance in accomplishing this. Before discussing this, however, we first consider the strategies that are currently available to the software designer.

1.2 Traditional Solutions

Modern programming languages such as Ada [10], Clu [54], Modula-2 [94], and Standard ML [61] provide data abstraction and encapsulation constructs called packages, clusters, or modules that enable one to define and enforce the boundaries separating the components of a software system. Modularity facilitates reuse and analysis and, when properly structured (either by design or through evolution), isolates and localizes the revisions that occur as a system is maintained, adapted, and reused [65]. In this paper, I refer to these data abstractions as *modules*. Modules can be viewed as implementing a kind of (usually complex) datatype definition. Like datatype definitions, there are several aspects to modules. These are the *abstract interface*, that is, the exported types and signatures of the operations; the underlying *representations* for the data objects created and manipulated by the module; and the *implementations* of the operations.

The integration of modules in a large-scale system is difficult. Modules that interact must agree not only on the abstract interfaces, but also on data representations in the cases where they share data. Also, as a software system evolves, the need to adapt existing interfaces can arise [66]. Thus, this problem of integration persists for as long as the system is maintained. Because the data representations affect the interactions among system components, I am motivated to use the term *module interface* to refer collectively to a module's abstract interface and associated data representations.

The Existing Choices in Software Development. Confronted with the problem of integrating interfaces in larger-scale systems, the software designer has the following choices:

1. Make an *a priori* correct choice of abstract interface and data representation definitions that will suffice for all anticipated needs.

   The UNIX system integrates tools using streams as a common interface and sequences of characters as a common data representation. Traditional database systems also devise common interfaces and data representations at the beginning of the design process. However, it is often the case that good data representations may be difficult to design *a priori*, especially when there is not much experience in the particular application domain. Once built, systems also evolve as users desire additional

functionality which may not have been anticipated initially. Adapting components is usually difficult once design decisions are made and, indeed, the cost of implementing change often becomes unmanageable. For example, adding a tool in UNIX that uses a complex internal data structure would involve introducing an expensive translation between it and the common interface. These problems of risk have led software designers toward iterative and evolutionary models of development [11], but little advice is given on how to get from one stage to the next.

2. Introduce functions for translating between representations in the situations where the abstract interfaces agree but the data representations do not.

   Separately designed modules that share data may be used together by writing translation functions that convert from one module's representation for data objects to the other's. However, the efficiency cost in the overhead of mapping back and forth among modules may not be acceptable.

   Take, for example, the case in which two modules have been separately designed for matrix operations, one for computing inverses and another for computing determinants. Let us assume that the modules agree on abstract interfaces, in which there are operations to create a matrix and also to obtain the elements of a given matrix. The modules differ, however, on their data representations, perhaps for reasons of efficiency. To use the modules together it is necessary to write translation functions that map one matrix representation to the other. This can be done by obtaining the elements of a matrix from one module and then creating a matrix with those elements in the other module.

3. Use a very-high-level language with appropriate built-in high-level types.

   In this case data representations are not explicitly defined. Instead, design decisions regarding data representations are left to a compiler (*eg.*, SETL [79]). This means, however, that the performance of the implementation and expressiveness of the programming language are limited by the existing compiler technology. Furthermore, if a designer wants to develop a system using rich abstractions that will have exacting performance requirements, then it seems that the designer must be involved in defining data representations.

4. Adapt or refine the abstract interfaces of existing modules by defining new modules as extensions of the existing ones.

   For example, object-oriented techniques can be used to define new types (and hence abstract interfaces) in terms of existing ones [55]. Objects having the new type will share meaning with objects of the existing type, typically by inheriting its operations and adding something more. The new objects will also share implementation by directly reusing the *code* for the existing objects. Unfortunately there is no formal way to specialize that implementation in the context of the new type in order to obtain better performance.

1.3 Focus of the Thesis

Each of these traditional approaches addresses the problem of module interface integration with varying degrees of success. I am interested in how program transformation might be used to complement or enhance them.

1.3.1 Thesis

Program transformations can provide systematic support for integrating general-purpose software modules into efficient systems. This approach also provides support for later adaptation.

In particular I am exploring the use of transformation-based techniques to (1) provide a systematic approach to adapting datatypes and modules, (2) remove the overhead of translation functions at runtime through program manipulation, (3) optimize the performance of datatypes using insight from the software developer, and (4) specialize implementations to obtain better performance in programs with modules that reuse code through inheritance. An evaluation of the utility of the techniques developed in this thesis is given in Chapter 7.

1.3.2 Approach of the Thesis Research

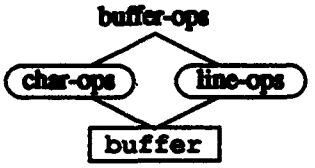
I use program-transformation methods to integrate module interfaces, yielding efficient implementations. Complex datatype definitions start as a collection of separate modules. Then, *translation functions* among the modules are introduced to reach preliminary (or "baseline") agreement on data representations, and *module extensions* defining new interfaces are used to reach agreement on abstract interfaces. The initial interfaces are then integrated and optimized by using an extended form of *datatype transformations*. This results in a single consistent and efficient implementation.

1.3.3 A Model of "Module Interface" Integration

Let us now consider the problem of designing and implementing a text buffer that manipulates characters and lines for a text editor. (This example will figure prominently in this thesis.) In a software engineering process, one of the early design issues, and one with the highest associated design risk, is the selection of the representation for a major data structure such as the buffer. After deciding on this representation (call it the `buffer`), it then remains to define the character and line operations, and finally the exported buffer operations. In the Standard ML [59] module system, for example, the program for buffer operations could be decomposed into two modules, one each for the character and line operations. The aim of modularization is to decompose programs into modules that can be manipulated relatively independently of each other.

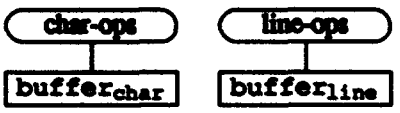
Standard ML handles the problem of sharing of data representations by providing a means for expressing and managing this interaction through its module system [42]. In this example, the two modules for character and line operations would be combined by using them as parameters to a "functor" (a parameterized module) that declares (via a "sharing

constraint" declaration) that they share a common buffer data structure, and then exports the buffer operations. (The sharing declaration is necessary, since without it, there may be a type conflict in the buffer operations module, since the buffer inherited through the character operations and the buffer inherited through the line operations would be interpreted as two distinct types.)



It is, unfortunately, usually difficult to make such *a priori* decisions on data representations, especially since new operations might be added after the initial design and implementation have long been completed.

An alternative approach is to define two separate modules, one for character operations and the other for line operations, each of which assumes its own specialized data representation for the `buffer`.



At some point these modules must be somehow integrated if we are to use both character and line operations on the same buffer. The problem of sharing is now more complex since the modules might not use the identical data representation for `buffer`. Rather, they may each define their own data representations, which are essentially "views" [29] of some "canonical" buffer. As indicated earlier, integration could be achieved by introducing functions that translate between the representations. (This would rely for consistency on an external unifying semantic-model.)

I propose a new approach. Rather than mediating the representations through translation functions at runtime (which likely incurs a significant performance penalty), these mappings are incorporated into a single buffer implementation by *deriving* a new common (and efficient) data representation. Program-transformation techniques provide a means to accomplish this by "synthesizing" a new data representation from the collection of specialized ones. In essence, the translation of interfaces is "shifted" to an earlier point in the computation (or "compiled"). An example is presented in Chapter 3.

This approach provides meaningful support for later adaptation. For example, suppose that at some future time the text buffer implementation is to be used in a new application, say, a display editor. The display editor may impose new requirements on the functionality of the buffer. In this case the buffer abstract interface must be extended. Such extensions could be accomplished by using, for example, inheritance (in the object-oriented sense) to restructure the interfaces and add the new functionality. Program-transformation techniques provide a means of fully integrating such extensions. In Chapter 4, I sketch out the inclusion of a finished text buffer in a display editor, deriving a new efficient implementation of the text buffer that takes advantage of the new context of the display editor.

1.3.4 What is New

Data transformations have been used to implement datatype specifications or to optimize existing programs. The approach of this thesis is novel in that the research extends data transformations by introducing transformation techniques for module interface design. This enables the application of transformation techniques to the development and adaptation of "larger-scale systems" [20]. Small-scale systems that deal primarily with algorithm development have been studied extensively. In this next step up I focus on data representations. This thesis does not address all issues of large-scale system design, but rather makes a contribution in extending the current techniques by applying program transformation to the integration of module interfaces. A study of the derivation of an interactive display-editor is shown that illustrates the hard problems of module interface integration that occur in software development. A framework for describing the transformation techniques is then established.

Solving this integration problem not only enhances existing approaches, but also may lead to new possibilities in designing systems. For instance, abstract datatypes are good for isolating clients from change, but not for promoting enhancement and adaptation [30]. This thesis research could lead to a new way of thinking about module construction where we imagine building more flexible systems that share data.

This thesis suggests a paradigm for datatype implementation by "components." Sometimes it is difficult to design types or anticipate future needs. Instead of introducing a type and anticipating all necessary operators, the operations are designed as we discover the need for them in the program using the datatype. The representations are selected based on the needed operations. This thesis also suggests a paradigm of system implementation by modules where we use transformation techniques to get better performance than simply reusing code. The module transformation system provides a way to manipulate the modules and to change the cohesiveness and the couplings of the modules [65].

1.4 Structure of the Thesis

I introduce the module interface transformation system in Chapter 2. First, background information on data transformation is presented that describes the progress made in developing data transformations. I continue the progression with techniques for datatypes and modules. The transformation system is introduced informally in two parts. First I enumerate a collection of module transformation rules that are useful for adjusting interfaces and data representations. Then I demonstrate how the rules are used in different strategies to implement datatypes by components, and to increase the efficiency of module systems.

In the two chapters that follow, I present an example that is centered around the derivation of an interactive display-editor to demonstrate the derivation process and techniques. In Chapter 3, I design an editor text buffer to illustrate the integration and adaptation of components to implement datatypes. Then in Chapter 4, I add a display to the buffer to illustrate the building of modules from other modules and to show how to increase the efficiency of the system.

I interpret the results of the editor derivation in Chapter 5 with a discussion of the alternatives in integrating the module interfaces that are available to the software designer at the design level and the implementation level. Also discussed are the criteria that the software designer might use to chose a specific alternative, the cost of the various alternatives, and the potential for scaling.

After the example, I examine the module interface transformation system in Chapter 6 to establish a framework for describing the derivation process and techniques. Providing a framework enhances the understanding of the terms used informally, provides structure to aid the software designer in using the approach and is an important step towards automating the system. Many of the terms that are introduced in the example, such as "component," "translation function," and "aggregate" are given a precise meaning. The decision was made to split the description of the module transformation system into two chapters in order to better motivate the system with an example before going into the technical details.

In Chapter 7, I evaluate the module transformation system by comparing it to traditional approaches and current research in software development, and then examine the applications in which the techniques would prove useful.

Finally, in Chapter 8, I conclude the thesis with a summary and evaluation of the contributions of the thesis. This thesis has explored a module interface transformation system without mechanized support which makes it difficult to apply the techniques to larger systems. Solutions to optimize the derivation process and to build an automated system are discussed.

Appendix A summarizes the notation used in the examples. Appendix B is a glossary of common terminology. The remaining appendices include details about the derivation and proof steps.

Chapter 2

Module Interface Transformation System

The general strategy for constructing systems using the module interface transformation system is illustrated in Figure 2.1. The process starts with the *design* of the top-level aggregate specification. The software designer who wishes to design a complex system is able to decompose the problem into components that best model that portion of the problem. This aggregate specification is used in the *integration* phase to produce an aggregate definition. The aggregate definition is in a format upon which data translations can be performed to obtain an executable *prototype*. Then additional transformations such as *expose*, *incorporate*, *release*, and *shift* can be performed to optimize the prototype into an efficient *implementation*. Later on the software designer may wish to introduce additional functionality in the *adapt* phase. The *design*, *integrate*, and *adapt* phases are supported by the methodology developed in Section 2.1. The *prototype* and *implement* phases are supported by the module transformation rules developed in Section 2.3. These phases will be elaborated as we progress through this chapter.

We begin in Section 2.1 which demonstrates how the module transformations are used in different strategies to implement datatypes by components and to increase the efficiency of module systems. To give us the necessary background, Section 2.2 contains information on data transformations that is necessary to understand the module transformation rules. Previous work on data transformation follows a progression of increasing support for larger-scale systems. Initial transformations affected the data representations of the parameters of functions; later transformation techniques were applied to abstract datatypes. The progression continues in Section 2.3 where I introduce applying transformation techniques to module systems, and enumerate the transformation methods on abstract interfaces and data representations. (A framework for describing the transformations is presented in Chapter 6.) Section 2.4 ties together the previous sections where I describe how these integration methods and the module transformation rules are used in the software development process.

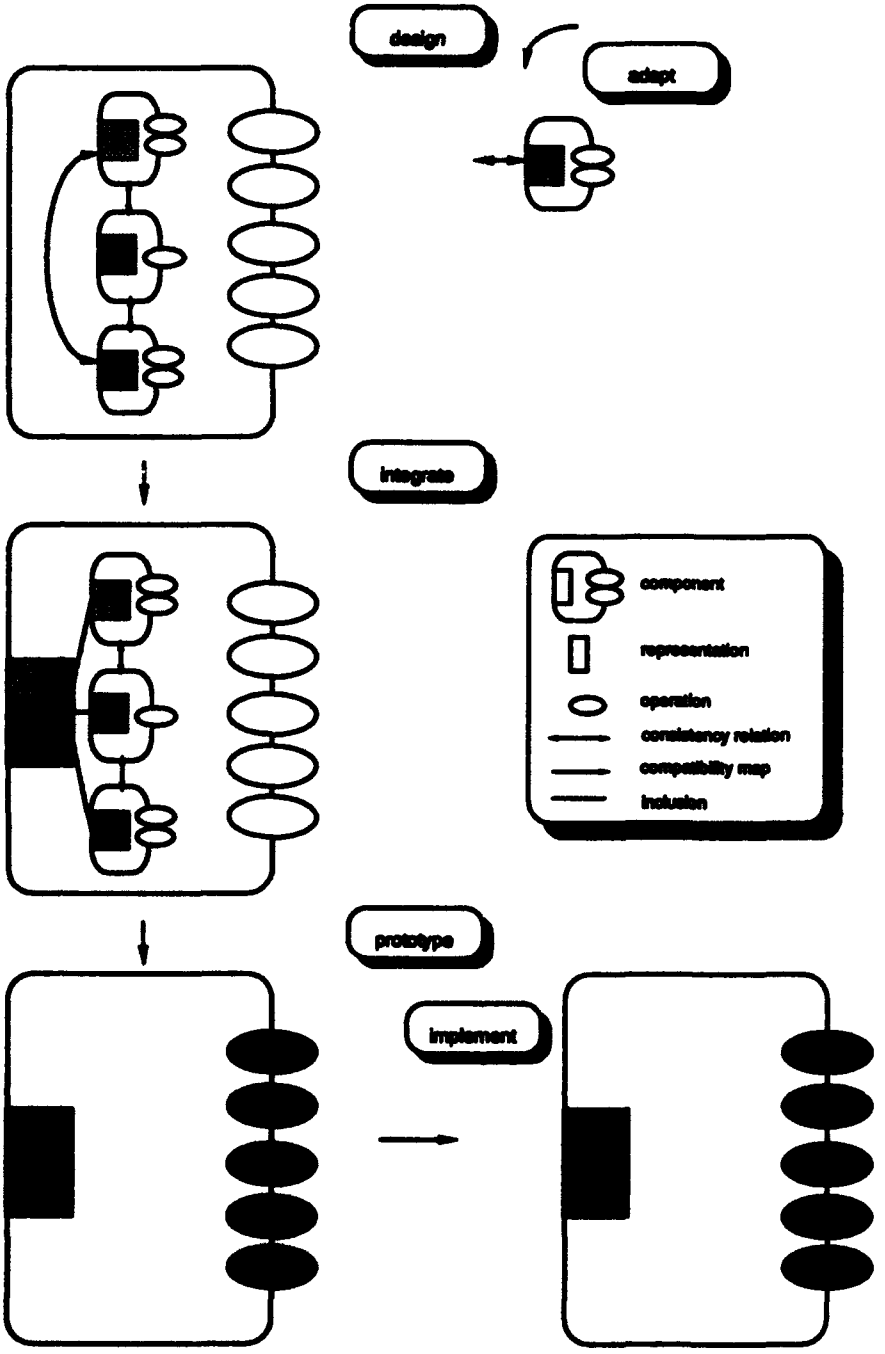


Figure 2.1: Strategy for Constructing Systems

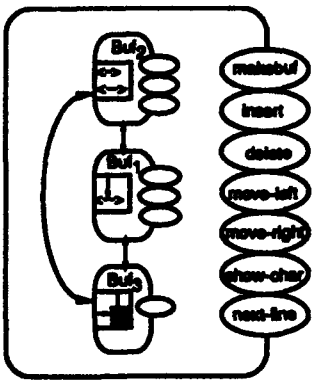


Figure 2.2: Buffer Definition

2.1 Module Transformation in Software Development

We have briefly seen in Section 1.3.3 an approach to the derivation of an editor buffer that is part of the development of an interactive text editor, and in Figure 2.1 a general strategy for constructing systems using this approach. Using this strategy, the editor is designed as a collection of separate modules, each of which implements subsets of the buffer operations efficiently (Figure 2.2). There are three modules: $\texttt{Buf}_1$ representing a buffer as a sequence of characters with an explicit index for the point where editing takes place; $\texttt{Buf}_2$ representing a buffer as a pair of sequences, corresponding to the characters to the left and to the right of the point of editing; and $\texttt{Buf}_3$ representing a buffer as a sequence of lines with an index consisting of a line and character position for the point of editing. *Compatibility maps* are introduced to establish a correspondence among the data representations. This collection of modules is then *integrated*. Using an extended form of data transformations (developed in the following sections) yields an executable *prototype* (*i.e.*, the first executable system); further specialization then yields a more efficient *implementation*.

The following two sections describe these techniques for integrating datatypes (Section 2.1.1) and modules (Section 2.1.2). Each section includes a description of the method, an example, and a pointer to how the method is used in the editor example in the following two chapters. The examples are extracted from the display-editor derivation in the two chapters that follow. On first reading it is best to skim through the example. The intention here is to show the "structure" of the method. Details about the notation, the display editor, and the method will be learned when we return to the examples in the editor derivation. We also return to these transformation methods and see a more formal definition in Chapter 6.

Notation and Naming Conventions. Names of types and operations have the name of the component in which they were defined prepended in order to resolve ambiguities. For example, if the components define a type `buf` then $\texttt{Buf}_1$`.buf` refers to the type defined in the first component. To enhance the readability of the examples, these "qualified"

names are abbreviated by omitting the component name, and using the component number as a subscript with the type or operation name. For example, $\texttt{buf}_1$ and move-right_1 are abbreviations for $\texttt{Buf}_1\texttt{.buf}$ and $\texttt{Buf}_1\text{.move-right}$. The expression "**local**...**in**...**end**" is used to separate the public operations (that appear after the keyword **in**) from the private operations (that appear after the keyword **local**) which are used by the public functions but not intended for use elsewhere. It is advisable for the reader to look first at the public functions and then at the private functions if additional details are desired.

2.1.1 Datatypes

The text buffer example suggests a paradigm for datatype implementation by "components." The building blocks for the paradigm are components that implement parts of a type. Sometimes it is difficult to design types or anticipate future needs. Instead of introducing a type and anticipating all necessary operators, the operations are designed as we discover the need for them in the program using the datatype. Here, we use modules to implement components.

Integrating a Collection of Components

We say what we mean by aggregating a number of components into a composite data structure by defining an *aggregate specification*. First we must supplement our notation (based on Standard ML [42]) with axioms to describe the properties of the data aggregate. Extended ML [72] gives the developer this ability to add axioms to structures and signatures. Axioms are expressions of boolean type and are built using the functions of first order logic.

For the purposes of this definition, we consider a system consisting of two components (which "represent" the same object), and consider the case where an operation is defined in the second component. The consistency relations, map_i^j, ensure the consistency among these components. A projection, proj_i, of an aggregate data object yields an object of a particular component, and each such component object is related to other component objects by a consistency relation. The following two axioms describe the properties of the aggregate.

$$\textbf{axiom}\ \text{proj}_2^1(\text{op}(\text{agg})) \;=\; \text{op}_2(\text{proj}_2^1(\text{agg}))$$

$$\textbf{axiom}\ \text{proj}_2^1(\text{agg})\ \ \text{map}_2^1\ \ \text{proj}_1^1(\text{agg}) \;\Longrightarrow\; \text{proj}_2^1(\text{op}(\text{agg}))\ \ \text{map}_2^1\ \ \text{proj}_1^1(\text{op}(\text{agg}))$$

The first axiom defines the behavior of the operation on the aggregate datatype induced by the second component operation, op_2. The remaining axiom ensures that after applying the operation, all of the components remain consistent. There is a set of such axioms for each operation defined. Only one set is shown for illustrative purposes.

Now that we have established what is meant by aggregating a number of components, we define how the aggregate is implemented. Given a collection of components and compatibility maps, an *aggregate definition* is produced that is amenable to module transformations (Figure 2.3). The compatibility maps, $\text{map}_{i\ldots j}$, respect the consistency relations and define how to make the components consistent. The function span uses the compatibility maps

Given: op_2, $map_{2\to 1}$
local
$span(c_2) \Leftarrow Agg(c_1, c_2)$ **where** $c_1 = map_{2\to 1}(c_2)$
in
$\underline{op}(span(c_2)) \Leftarrow span(op_2(c_2))$
end

Figure 2.3: Aggregate Definition

to translate between the component and the data aggregate. The operation, op, on the data aggregate can thus be defined using span in a form amenable to transformation. Since more than one operation may occur on the lefthand side of an expression-procedure definition, there may be some confusion about which operation is being defined. Because of this, the operation being defined are underlined to distinguish it from the others. This is but one example of a definition given a certain set of translation functions. The general case is treated in Section 6.1.

The Steps of Integration. The essential steps for implementing datatypes by components:

1. Specify the overall datatype interface. The names of the operations are listed with their signatures.
2. Define the component implementations, each of which implements some subset of the overall interface. Collectively, all of the components implement the entire interface.
3. Define functions that translate from one component into another to establish the consistency of the collection of data representations.
4. Choose the product of the component representations as an expedient representation.
5. Integrate the components to define the composite datatype. Each operation defined in a component induces a corresponding operation on the aggregate datatype. Each aggregate datatype operation is defined (in a form amenable to transformation) in terms of the component where the operation was defined.

Example. Suppose we are given two components that implement different buffer operations and a relation that defines what it means for them to be consistent. The operations move-right and show-char are defined in the first component, $\texttt{Buf}_1$, makebuf is defined in the second component, $\texttt{Buf}_2$. The consistency relation is defined by map_1^2. Using the axioms for the aggregate specification as our template, we combine these two components into a composite data structure.

axiom $\text{proj}_1(\text{makebuf}) = \text{makebuf}_2$
axiom $\text{proj}_1(b)\ \text{map}_1^2\ \text{proj}_2(b) \Longrightarrow \text{proj}_1(\text{makebuf})\ \text{map}_1^2\ \text{proj}_2(\text{makebuf})$
axiom $\text{proj}_1(\text{move-right}(b)) = \text{move-right}_1(\text{proj}_1(b))$
axiom $\text{proj}_1(b)\ \text{map}_1^2\ \text{proj}_2(b) \Longrightarrow \text{proj}_1(\text{move-right}(b))\ \text{map}_1^2\ \text{proj}_2(\text{move-right}(b))$
axiom $\text{show-char}(b) = \text{show-char}_1(\text{proj}_1(b))$

Using an algorithm that preserves the properties of the axioms, an aggregate definition is produced from the axioms. This algorithm is explained in Section 6.1.4.

Structure `Buf` : BUF = **struct**
  structure $\text{Buf}_1, \text{Buf}_2$
  type `buf` = `Buf` **of** (`int` × `ch`* × `ch`* × `ch`*)
    local
      $\text{map}_{2\to 1}(\text{Buf}_2(l,r)) \Leftarrow \text{Buf}_1(\#l, l \mathbin{@} r)$
      $\text{span}(\text{Buf}_2(l,r)) \Leftarrow \text{Buf}(p, t, l, r)$ **where** $\text{Buf}_1(p,t) = \text{map}_{2\to 1}(\text{Buf}_2(l,r))$
      $\text{unspan}(\text{Buf}(p,t,l,r)) \Leftarrow \text{Buf}_1(\{ p \mid p_2 \}, \{ t \mid t_2 \})$
        where $\text{Buf}_1(p_2, t_2) = \text{map}_{2\to 1}(\text{Buf}_2(l,r))$
    in
      $\text{makebuf} \Leftarrow \text{span}(\text{makebuf}_2)$
      ...
      $\text{unspan}(\text{move-right}(b)) \Leftarrow \text{move-right}_1(\text{unspan}(b))$
      $\text{show-char}(b) \Leftarrow \text{show-char}_1(\text{unspan}(b))$
    end
  end

Notice how the definitions for makebuf, move-right, and show-char are in the form of data transform procedures from the template in Figure 2.3. The notation $\{ v_1 \mid v_2 \}$ used in the definition of unspan denotes alternative ways of computing the same value. It is necessary in this case to preserve the consistency between the data structures of the two components that comprise the data aggregate. Sometimes unspan is used instead of span depending upon what translation functions are available.

Using the Transformation. This integration process is useful for integrating collections of components to implement datatypes. A detailed example is shown in Section 3.1.

Adding a New Component

We adapt the system by introducing a new component, c_3, which defines the operation, x. The aggregate specification for the new system is defined by supplementing the previous definition for the aggregate specification with a collection of axioms defining the operations induced from the new component. The first axiom defines the behavior of the operation on the aggregate datatype in terms of the new component. The following three axioms ensure that after applying the operation, all of the components are kept consistent. The last two axioms are extensions to the previously defined axioms for op. They ensure that the newly introduced component is kept consistent after applying an operation in the existing system.

Given: op_2, $map_{2\to 1}$, x_3, $map_{2\to 3}$

local

$\quad unspan(Agg_{2\times 3}(c_2, c_3)) \Leftarrow$
$\quad\quad \{ map_{2\to 3}(c_2) \mid c_3 \}$
$\quad span(Agg_{2\times 3}(c_2, c_3)) \Leftarrow$
$\quad\quad Agg'(c_1, c_2, c_3)$ **where** $c_1 = map_{2\to 1}(c_2)$
$\quad span'(Agg(c_1, c_2)) \Leftarrow$
$\quad\quad Agg'(c_1, c_2, c_3)$ **where** $c_3 = map_{2\to 3}(c_2)$

in

$\quad \underline{unspan}(\underline{x}_{2\times 3}(Agg_{2\times 3}(c_2, c_3))) \Leftarrow$
$\quad\quad x_3(unspan(Agg_{2\times 3}(c_2, c_3)))$
$\quad \underline{x}(span(Agg_{2\times 3}(c_2, c_3))) \Leftarrow$
$\quad\quad span(x_{2\times 3}(c_2, c_3))$
$\quad \underline{op'}(span'(Agg(c_1, c_2))) \Leftarrow$
$\quad\quad span'(op(Agg(c_1, c_2)))$

end

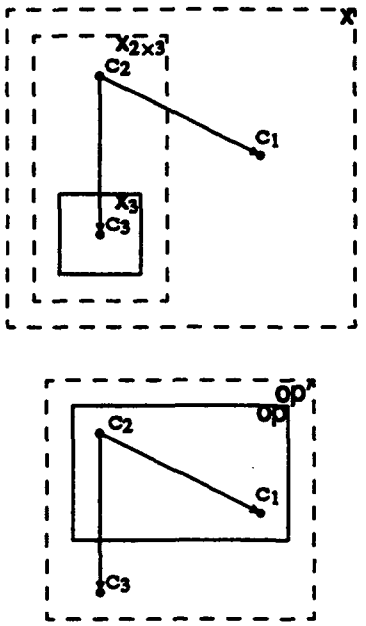


Figure 2.4: Merging with the Original System

axiom $proj_3(x(agg)) = x_3(proj_3(agg))$

axiom $proj_2(agg)\ map_2^1\ proj_1(agg) \Longrightarrow proj_2(x(agg))\ map_2^1\ proj_1(x(agg))$
axiom $proj_3(agg)\ map_3^1\ proj_1(agg) \Longrightarrow proj_3(x(agg))\ map_3^1\ proj_1(x(agg))$
axiom $proj_3(agg)\ map_3^2\ proj_2(agg) \Longrightarrow proj_3(x(agg))\ map_3^2\ proj_2(x(agg))$

axiom $proj_3(agg)\ map_3^1\ proj_1(agg) \Longrightarrow proj_3(op(agg))\ map_3^1\ proj_1(op(agg))$
axiom $proj_3(agg)\ map_3^2\ proj_2(agg) \Longrightarrow proj_3(op(agg))\ map_3^2\ proj_2(op(agg))$

There is some flexibility in how to integrate the new component to produce the aggregate definition. Here, we consider four alternative methods of component adaptation. These definitions all satisfy the axioms. We have the choices to "merge" or "translate" the new component with the components in the original system or with the data aggregate implementation.

Merging with the Original System. We could merge the new component with the components in the original system (see Figure 2.4). The first two public definitions (after the keyword **in**) incrementally build a definition for the new operation (using the method introduced in Figure 2.3). The first definition "spans" the representation from the new component to an intermediate aggregate consisting of the second and third component, $Agg_{2\times 3}$. The next definition "spans" the representation from this intermediate aggregate to the new data aggregate consisting of all three components. Two steps are necessary in this case because of the translation functions available. Since we are not given a function that translates from the third component to the first component directly, we must first merge the third component with the second component, and then with the first component. Compare

Given: op_2, $map_{2\to 1}$, x_3, $map_{2\to 3}$

local

$unspan(Agg_2(c_2)) \Leftarrow map_{2\to 3}(c_2)$

$span(Agg_2(c_2)) \Leftarrow Agg(c_1, c_2)$ **where** $c_1 = map_{2\to 1}(c_2)$

in

$unspan(\underline{x}_2(Agg_2(c_2))) \Leftarrow x_3(unspan(Agg_2(c_2)))$

$\underline{x}(span(Agg_2(c_2))) \Leftarrow span(x_2(c_2))$

end

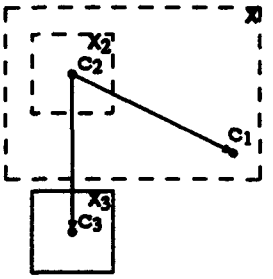


Figure 2.5: Translating into the Original System

these two definitions with the definition of integrating the existing system in Figure 2.3. The diagram to the top-right in the figure illustrates the process. The labeled nodes represent the components which are connected by directed arcs that represent the compatibility maps. The solid box encloses the component where the operation of interest is defined, in this case, component c_3 where x_3 is defined. Dashed boxes represent the derived operations on the intermediate and final aggregate (there is one for each of the two definitions). Examine them starting at the inner-most box and proceeding outward. First $x_{2\times 3}$ is derived (corresponding to the first definition) and then x.

We also add the last definition to extend the previous definitions for the operations in the existing system. We start with the definition of op defined in the aggregate consisting of c_1 and c_2 (solid box in the bottom-right diagram); then we derive op′ to include c_3 (outer dashed box).

Translating into the Original System. An alternative to merging is translating the new component into the existing system (Figure 2.5). Only the operations of the new component need to be defined, since the existing system does not change. As with merging, the two public definitions incrementally build a definition for the new operation. The first definition "spans" the representation from the new component to the existing component, rather than the intermediate aggregate in Figure 2.4, since we are translating and not merging. The next definition "spans" the representation from the second component to the data aggregate of the existing system. There is no third definition, as was the case in merging, because the existing operations do not have to be modified since the aggregate representation remains the same.

Merging with the Implementation. Rather than starting with the original definitions of the existing system, at times it may be beneficial to treat the implementation of the aggregate as a component, and merge the aggregate and new component directly. Recall that the process of obtaining an implementation from the aggregate definition involves applying transformations to the definition to obtain a prototype, and then specializing the prototype to obtain an implementation. This specialization process can be characterized as a translation function, $map_{p\to i}$, that maps prototypes into implementations. For the sake

Given: x_3, $map_{i_2 \to 3}$

local

$unspan(Agg_{i_2 \times 3}(i_2, c_3)) \Leftarrow \{ map_{i_2 \to 3}(i_2) \mid c_3 \}$

$span(Agg_{i_2 \times 3}(i_2, c_3)) \Leftarrow Agg_{i_1 \times i_2 \times 3}(i_1, i_2, c_3)$ **where** $i_1 = map_{i_2 \to i_1}(i_2)$

$span'(Agg(i_1, i_2)) \Leftarrow Agg'(i_1, i_2, c_3)$ **where** $c_3 = map_{i_2 \to 3}(i_2)$

in

$unspan(x_{i_2 \times 3}(Agg_{i_2 \times 3}(i_2, c_3))) \Leftarrow x_3(unspan(Agg_{i_2 \times 3}(i_2, c_3)))$

$\underline{x}(span(Agg_{i_2 \times 3}(i_2, c_3))) \Leftarrow span(x_{i_2 \times 3}(i_2, c_3))$

$\underline{op'}(span'(Agg(i_1, i_2))) \Leftarrow span'(op(Agg(i_1, i_2)))$

end

Figure 2.6: Merging with the Implementation

of concreteness, let us say that the specialization process takes the two components of the prototype aggregate, and specializes the first component and keeps the second component intact.

$$map_{p \to i}(Agg(c_1, c_2)) \Leftarrow Agg_i(f(c_1), c_2)$$

Perhaps we can define a translation function, $map_{i_2 \to i_1}$, for the relationship between the two components in the implementation in terms of the compatibility map, $map_{2 \to 1}$ (which expresses the relationship between the components in the prototype), and the translation function, $map_{p \to i}$ (which expresses the relationship between the prototype and the implementation). Then, if we define the relationship between the new component and the implementation as the translation function, $map_{i_2 \to c_3}$, we obtain a definition for the adapted system (Figure 2.6). Notice the similarities between this definition and the previous one in Figure 2.4. We have substituted $map_{i_2 \to i_1}$ for $map_{2 \to 1}$ which "promotes" the specialization filter (*i.e.*, f in $map_{p \to i}$) to the time the integration is performed so that we may be spared doing extra work only to eliminate it later in the derivation.

We are able to treat the implementation of the aggregate as a component only under certain conditions though. When there are no interdependencies among the aggregate, then it is simple to treat the aggregate as a component, since there are no "internal" translation functions to be concerned with. When the translation functions are many-to-one, it may not always be possible.

Translating into the Implementation. An alternative to merging is translating the new component into the existing system (Figure 2.7). Only the operations of the new component need to be defined, since the existing system does not change. The same comparison made between merging and translating on the original system applies to merging and translating on the implementation.

The Steps of Adaptation. Adaptation, adding a new component, is similar to the original problem of designing and implementing the initial system, since both tasks can be conceptualized as the task of integrating components.

Given: x_3, $map_{i_2 \to 3}$

local

$unspan(Agg_{i_2}(i_2)) \Leftarrow map_{i_2 \to 3}(i_2)$

$span(Agg_{i_2}(i_2)) \Leftarrow Agg_{i_1 \times i_2}(i_1, i_2)$ **where** $i_1 = map_{i_2 \to i_1}(i_2)$

in

$unspan(x_{i_2}(Agg_{i_2}(i_2))) \Leftarrow x_3(unspan(Agg_{i_2}(i_2)))$

$x(span(Agg_{i_2}(i_2))) \Leftarrow span(x_{i_2}(i_2))$

end

Figure 2.7: Translating into the Implementation

The steps for adapting a datatype by adding a new component:

1. Extend the overall datatype interface. The names of the new operations and type information are added to the signature.
2. Define the component implementation, using a data representation most suitable for the operations.
3. Write a single translation function between this component and another in the existing system.
4. Choose the product of the new component representation and component representations of the existing system as an expedient representation. Or, as an alternative choice, keep the representation of the existing system.
5. Integrate the new component by adding new definitions for the operations defined in the new component, and by extending the definitions for the operations in the existing system to update the new component. (If the representation of the existing system is chosen, then the operations of the existing system do not have to be redefined.) Each aggregate datatype operation can be defined (in a form amenable to transformation) in terms of the component where the operation was defined.

Example. We use the *merging with the original system* approach in this example. Recall that the buffer system in the example of the previous section was defined using axioms that specified how to integrate the original two components. This definition is enriched to include a new component for pages by adding these axioms:

axiom $proj_p(\text{forward-page}(b)) = Buf_p.\text{forward-page}(proj_p(b))$

axiom $proj_1(b)\ map_1'\ proj_p(b) \Longrightarrow proj_1(\text{forward-page}(b))\ map_1'\ proj_p(\text{forward-page}(b))$

axiom $proj_2(b)\ map_2'\ proj_p(b) \Longrightarrow proj_2(\text{forward-page}(b))\ map_2'\ proj_p(\text{forward-page}(b))$

The first axiom specifies the behavior of the forward page operation on the data aggregate in terms of the page component where it was defined. The remaining axioms ensure that the system remains consistent after the operation is performed.

Additionally, the axioms for the original operations must be extended to ensure consistency of the new component when an old operation is applied. For example, the axioms for move-right would be extended with,

$$\textbf{axiom}\ \mathsf{proj}_1(b)\ \ \mathsf{map}_1^p\ \ \mathsf{proj}_p(b) \implies \mathsf{proj}_1(\text{move-right}(b))\ \ \mathsf{map}_1^p\ \ \mathsf{proj}_p(\text{move-right}(b)).$$

In order to produce an aggregate definition, a compatibility map that respects map_1^p must be provided. In the compatibility map that follows, the new page component is computed from the original $\mathtt{Buf}_1$ component using the auxiliary functions npages to count the number of pages in the sequence of characters to the left of the cursor (where $s\,[..i]$ is the subsequence of s from the beginning to i), and chars2pages to parse the sequence of characters into a sequence of pages.

$$\mathsf{map}_{1\to p}(\mathtt{Buf}_1(p,t)) \Leftarrow \mathtt{Buf}_p(\text{npages}(t\,[..(p-1)]), \text{chars2pages}(t))$$

We obtain a new definition for each page operation in the new data aggregate (using an extension of the algorithm that integrated the original system in the previous example). For example, the definition for forward-page follows.

$$\begin{array}{l}
\textbf{local} \\
\quad \text{unspan}(\mathtt{Buf}_{1\times p}(p,t,pi,tp)) \Leftarrow \{\, \mathsf{map}_{1\to p}(\mathtt{Buf}_1(p,t)) \mid \mathtt{Buf}_p(pi,tp)\,\} \\
\quad \text{unspan}'(\mathtt{Buf}'(p,t,l,r,pi,tp)) \Leftarrow \mathtt{Buf}_{1\times p}(\{\,p \mid p_2\,\},\{\,t \mid t_2\,\},pi,tp) \\
\qquad\qquad\qquad\qquad \textbf{where}\ \mathtt{Buf}_1(p_2,t_2) = \mathsf{map}_{2\to 1}(\mathtt{Buf}_2(l,r)) \\
\textbf{in} \\
\quad \underline{\text{unspan}(\text{forward-page}_{1\times p}(b))} \Leftarrow \text{forward-page}_p(\text{unspan}(b)) \\
\quad \underline{\text{unspan}'(\text{forward-page}(b))} \Leftarrow \text{forward-page}_{1\times p}(\text{unspan}'(b)) \\
\textbf{end}
\end{array}$$

Notice how this take two steps. First we obtain a definition for forward-page for the intermediate aggregate $\mathtt{Buf}_{1\times p}$, and then for the new aggregate consisting of the product of all the components.

We also obtain a new definition for each operation of the existing components in the new aggregate. For example, the move-right operation must be extended to define the operation on the new aggregate that includes the new page component in terms of the old aggregate. This is done by simply adding a new definition.

$$\begin{array}{l}
\textbf{local} \\
\quad \text{span}(\mathtt{Buf}(p,t,l,r)) \Leftarrow \mathtt{Buf}'(p,t,l,r,pi,tp) \\
\qquad\qquad\qquad\qquad \textbf{where}\ \mathtt{Buf}_p(pi,tp) = \mathsf{map}_{1\to p}(\mathtt{Buf}_1(p,t)) \\
\textbf{in} \\
\quad \underline{\text{move-right}'(\text{span}(b))} \Leftarrow \text{span}(\text{move-right}(b)) \\
\textbf{end}
\end{array}$$

Notice how the definitions for forward-page and move-right compare with the template in Figure 2.4. We may substitute span for unspan or vice-versa depending on what compatibility maps are available.

Using the Transformation. This adaptation process is useful for adapting datatypes and introducing new functionality. A detailed example is shown in Section 3.2.

2.1.2 Larger-Scale Systems

This next example of using the text buffer as part of a display editor suggests a paradigm of system implementation by modules where we use transformation techniques to get better performance than simply reusing code. The building blocks for this paradigm include objects and modules (see [33] and [90] and others, [9], [58], [87]). Larger-scale systems can be built using modules hierarchies (*i.e.*, modules that import other modules). The module transformation system provides a way to manipulate these building blocks and to change the cohesiveness and the couplings of the modules. Initial experience suggests that techniques demonstrated for the integration of components are applicable to module systems as well.

Adapting Module Interfaces

New modules can be defined in terms of existing modules to adapt abstract interfaces. This is accomplished by defining a module that imports an existing module, using some of the existing functions, and perhaps adding additional functions. This is different from adaptation by adding a new component because in addition to adding new functions, we can delete or modify them as well.

We are able to build a system using a hierarchy of modules where data among the modules may be shared. Operations from the imported module can be "propagated" into the importing module. We say what is meant by extending a module by using axioms to define a specification. The single axiom below defines the behavior of the propagated operation op′ in the importing module in terms of the imported module operation op.

```
axiom proj(op'(a))  =  op(proj(a))
```

Now we define how this specification is implemented. Given the imported module and a function mapping between the data in the imported and importing module, $map_{i \to m}$, new definitions for the operations are defined.

```
local
   span(c)  ⇐  Agg(c', s)
                  where c' = map_{i→m}(c)
                     and s = f(c)
in
   op'(span(c))  ⇐  span(op(c))
end
```

The importing module, M', starts with the imported module, M, and adds something extra, perhaps a new data field or additional operations. In this example M' uses the data from M and adds a new field that is computed using f. The function span uses the mapping function to translate between M and M' so that op′ is defined using span as a data transform procedure.

The Steps of Adaptation. The steps for adapting module interfaces using module extensions is similar to the process for integrating components:

1. Specify the module interface. The names of the operations are listed with their signatures.
2. Define the implementation. This is done by applying this same methodology to define a new module, or by using or adapting a module from a reusable library of software components.
3. Choose a representation, possibly including other modules as substructures, and defining the data representation in terms of the datatypes contained in these modules.
4. Define the implementation in terms of the substructures. There may be a correspondence (*eg.*, data invariance) between: (1) two imported modules, or (2) a module and the module it imports. This correspondence is expressed as a translation function.

Example. We define a multiple-buffer display-editor using a module that defines the screen and a module that defines "generic" association list (`Alist`) operations to manage the state of the buffers. The display editor inherits the state from the `Alist` module and adds something more, the state of the screen. Each operation in the `Alist` module induces a corresponding operation in the display editor.

The `Alist` module has an operation select for selecting data in a list given a key. We use it to look up a buffer object given its name. This new buffer operation select-buffer satisfies the axiom:

$$\text{axiom proj(select-buffer}(n, a)) \quad = \quad \text{select}(n, \text{proj}(a))$$

An `Alist` is represented as a list of pairs (of keys and data), where the beginning of the list is cached. The selected data is moved to the beginning of the list in this cached position. The relationship between the `Alist` module and the display-editor module, `Ded-Mbsw`, that includes it can be expressed as a translation function.

$$\text{span}(\texttt{Alist}(l', k, d)) \quad \Leftarrow \quad \texttt{Ded-Mbsw}(\texttt{Alist}(l', k, d), s) \quad \text{where } o' = \text{policy}(d),\ s = \text{disp-to-screen}(o', d)$$

Notice how `Ded-Mbsw` uses the state from the `Alist` module and adds the state of the screen, s. The screen is computed from information contained in the `Alist` such as the current buffer and origin. The operations for the multiple-buffer display-editor are then defined in terms of the `Alist` operations.

$$\text{select-buffer}(n, \text{span}(a)) \quad \Leftarrow \quad \text{span}(\text{select}(n, a))$$

The operations defined in the `Alist` module are then reimplemented in the context of the display-editor module. For example, instead of select-buffer having to call select, a new definition for select-buffer is derived that operates on the state of the buffers directly. Not

only does this increase performance by eliminating the invocation of select, but it enables the possibility of further transformations that could specialize the screen in the context of the buffer state (*eg.*, incremental update).

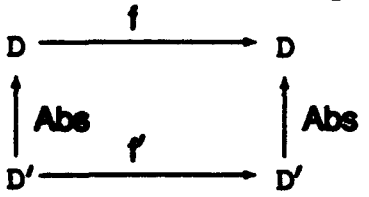
Keep in mind that this example demonstrates a particular set of design decisions. The software developer has in many cases, a range of alternatives to choose from to produce a variety of solutions.

Using the Transformation. This transformation provides a controlled methodology for propagating change and increasing the efficiency of module systems by tighter coupling. See Chapter 4 for a detailed example.

2.2 Data Transformation

Early data transformation methods focused on the relationship between abstract programs and their implementations. Hoare [46] presented a method for proving the correctness of a data representation for an abstract program.

Given an abstract program f on an abstract domain D, the concrete program f′ on the concrete domain D′ is a correct implementation of f if the following diagram commutes.

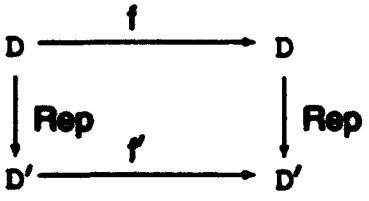


That is (where d' is an element of domain D′),

$$f(Abs(d')) = Abs(f'(d')). \qquad (1)$$

The abstraction function Abs maps the concrete items into the abstract objects which they represent. This approach has been adopted by VDM [7] where Abs is called a "retrieve" function.

An alternative approach is to derive the concrete representation using program transformation [12] rather than invent the concrete representation and then prove it correct. The representation function Rep maps the abstract object into a concrete representation. This direction is chosen to simplify the derivation but is only applicable when an injective representation function can be defined (though it does not necessarily have to be unique).



$$f'(Rep(d)) = Rep(f(d)) \qquad (2)$$

While the above equation may not immediately suggest an implementation, in many cases transformations can be applied to obtain an executable definition for f'. Darlington [17] shows how a concrete program is derived from an abstract program using program transformations, thereby ensuring that the implementation is correct; given an abstract program f on an abstract domain D and a target domain D', the concrete program f' and the mapping Rep can be derived. Of course it is not enough to transform only one function — all functions that operate on objects in the domain must be transformed. This can be more easily accomplished by using datatypes to group the functions that operate on an object.

Not only can transformations be applied to get from abstract to concrete programs, they can also be applied to concrete programs (or datatypes) themselves. Wile [91] develops this idea by considering the interrelationships along data paths in programs and outlining a set of informally described operations on datatypes. These include operations for delaying or advancing computation and operations for changing type signatures based on the "theory operations" of Burstall and Goguen [13].

Jørring and Scherlis [49, 76] develop and generalize these ideas even further to obtain a framework that permits programmers to take general-purpose abstract datatype definitions (which might come from a reuse library) and, using type transformations, obtain types tailored to the application. Described in terms of the above diagram, a given datatype D with its associated operations, for example, f, can be adapted to yield the specialized datatype D' with its associated operations, f'. The mapping Rep is also derived using the context in which the datatype appears.

2.3 Module Transformation Rules

Scherlis [76] uses four strategies called *incorporate*, *release*, *expose*, and *shift* for transforming abstract interfaces and data representations. These are techniques for adjusting interfaces and data representations within a given system. The *shift* strategy is a data-representation transformation that has a direct effect on program performance. Depending on the relative frequencies of operations, computation is moved between generation time (where information can be cached) and access time (where information can be computed on demand). The *incorporate* and *release* strategies transform abstract interfaces by moving abstraction boundaries (*i.e.*, internal program interfaces as defined by type signatures) to facilitate improvements in the efficiency of the type representations. The *expose* strategy transforms both data representation and abstract interfaces. The internal structure of data objects is revealed, moving the abstraction boundary of the type "inward," creating new abstract type names to move from more abstract representations to more concrete representations. An example of the transformation of a data representation is given in Section 3.1.5 (with details in Appendix C).

The module transformation rules described in this section start with Scherlis' framework. The intention is to have a small canonical set of transformations supplemented with knowledge about the domain of the data structure that are used to simplify the datatype. Each of the following subsections describes a module transformation rule and includes: an informal description of the transformation method, an example, and a pointer to how

the method is used in the editor example. The informal description and example show the end result of the transformation strategy, but not the intermediate steps. The intent is to present an overview of the transformations to quickly reach Section 2.4 where they are used in constructing software systems. Different languages are used in the examples to demonstrate that the transformations are not restricted to a given language but to languages that support module systems in general. After the editor example is presented, we return to these transformation methods in Chapter 6 where we see the steps involved in applying the transformation and a more formal definition.

When first reading this section, it is sufficient to examine the introductory prose, the example, and the pointer to how the method is used in the extended example. After reading the extended example, the reader may wish to return to this section to look at the descriptions to learn more of the details.

Notation and Naming Conventions. The interface of the datatype is represented by the type name and the signatures of the operations. Here is a template, where the italicized variables are to be filled in.

Interface *d* **is**
  $\{f : t\}^*$
end

The interface of the datatype d consists of the functions (represented by) f with the type t. One or more instances of a pattern is denoted using $\{\ldots\}^*$.

The implementation of the datatype is represented by the type name, the data representation and the implementations of the operations.

Rep *d* **is** α
  with
    $\{f(v^*) \Leftarrow b\}^*$
  end

The implementation of the datatype d consists of the data representation α, and the implementations of the functions f with formal parameters v^*, and bodies b. Function definition is denoted by $\Leftarrow$. When dealing with abstract datatypes (as opposed to modules), Rep and Abs are used to manage abstraction boundaries as in the original ML [35]. Rep reveals the underlying data structure of an abstraction, Abs creates an abstraction. Rep appearing on the righthand side of a procedure definition has the same effect as Abs appearing on the lefthand side, revealing the underlying data structure of an abstraction.

A representative selection of the operations are shown in the descriptions. For our purposes, the operations are categorized in terms of whether they produce an instance of the type, operate on the type, or reveal some information about the type. These are called generators, extensions, and observers, which are given the names, gen, ext, obs. The implementation of the operation is signified as a pattern, using G, E, or O for the three representative operations (*eg.*, ext(Abs(a)) $\Leftarrow$ Abs(E(a))). Although these categories correspond to the ones used in defining algebraic specifications of abstract datatypes,

the reason to use them here is to demonstrate the applicability of the transformational techniques.

The `typewriter` font is used for datatype names, sans-serif font for function names, and *italics* for variable names. The product type constructor $\times$ binds more tightly than the function type constructor $\rightarrow$.

2.3.1 Translate

Using the *translate* transformation, the software developer can change the representation of a datatype and/or move computation along the data paths of a program. This change in representations is expressed by a function that maps from the original representation into the new one. The transformation provides a mechanical means (with guidance from the user) to reimplement the operations of this datatype on the alternative data representation. The meaning of the abstract datatype, however, remains the same.

Example. A simple buffer datatype has operations to create a new instance of the buffer, insert a character, and show the character at the point of editing. Standard ML [60] notation is used to represent this datatype; the abstract interface is denoted by an ML signature.

```
Signature BUF = sig
    type buf
        val makebuf : buf
        val insert : ch * buf -> buf
        val show-char : buf -> ch
    end
```

The implementation of the datatype is denoted by an ML structure. A buffer with a data structure that represents the buffer as a pair of sequences of characters (@ is append),

```
Structure Buf : BUF = struct
    type buf = Buf of (ch List * ch List)
        val makebuf = Buf(nil, nil)
        fun insert(c, Buf(l,r)) = Buf(l @ list(c), r)
        fun show-char(Buf(l,r)) = last(l)
    end
```

and a function that translates this data representation into a new one consisting of an index and a sequence of characters,

```
span(Buf(l,r)) = Buf'(len(l), l @ r)
```

are transformed (after several transformation steps). The new data structure represents the buffer as an index to the point of editing and text.

```
Structure Buf' : BUF = struct
    type buf = Buf of (int * ch List)
        val makebuf  =  Buf(0, nil)
        fun insert(c, Buf(p,t))  =  Buf(p+1, subseq(t,0,p-1) @ list(c) @ subseq(t,p,len(t)))
        fun show-char(Buf(p,t))  =  nth-item(t, p-1)
    end
```

A different example with the steps included is shown in Appendix C.

Description. We illustrate the transformation process with a general datatype. The abstract interface for the general datatype, `Dtype`, containing a representative collection of operations follows.

```
Interface Dtype is
    gen :  Dtype
    ext :  Dtype → Dtype
    obs :  Dtype → β
    end
```

The operations are defined on a datatype with representation α. For example, ext takes an instance of the datatype as an argument, reveals the underlying data representation of the abstraction (represented by a), performs some operations on it (represented by the pattern E), before returning the result as a new abstraction.

```
Repn Dtype is α
    with
        gen  ⇐  Abs(G)
        ext(Abs(a))  ⇐  Abs(E(a))
        obs(Abs(a))  ⇐  O(a)
    end
```

The function span that maps the given datatype into the new representation is provided below. The translation of the data representation is performed by S. The new abstraction boundary, Abs', is primed to distinguish it from the old, but they serve the same purpose, that is, to ensure that the underlying data representation remains hidden outside of the type. The span function is special then, since it is can manipulate these abstraction boundaries.

```
S :  α → α'
span(Abs(a))  ⇐  Abs'(S(a))
```

Applying the *translate* transformation produces new implementations of the operations defined on the data representation α'. The steps are shown in Figure 2.8. Typically, the transformation proceeds where: (1) the bodies of the old operation and span function are expanded; (2) domain knowledge about the data representation is applied to simplify the body; (3) an insight step is applied to "bridge" the old and new representations; (4) additional simplification steps are applied; and (5) the span function is abstracted from the body of the operation. More information is provided in Section 6.2.1.

$f'(\text{span}(A)) \Leftarrow \text{span}(f(A))$
$f'(\text{span}(A)) \Leftarrow S(F(A))$
$f'(\text{span}(A)) \Leftarrow B(A)$
$f'(\text{span}(A)) \Leftarrow B'(A)$
$f'(\text{span}(A)) \Leftarrow F'(S(A))$
$f'(\text{span}(A)) \Leftarrow F'(\text{span}(A))$
$f'(A) \Leftarrow F'(A)$

Figure 2.8: Transformation Steps

Repn Dtype **is** α'
with
gen $\Leftarrow$ $\text{Abs}'(G')$
$\text{ext}(\text{Abs}'(a)) \Leftarrow \text{Abs}'(E'(a))$
$\text{obs}(\text{Abs}'(a)) \Leftarrow O'(a)$
end

Thus, given the initial datatype, a function that maps the old representation into the new representation, and some insight from the software developer to guide the transformation, the new datatype is produced. The datatype consists of the new implementations of the operations, represented by G', E', O', that operate on the new representation.

Using the Transformation. The translate transformation is used for optimization or integration and implementation. Used as an alternative to *shift*, computation can be moved along data paths to increase the efficiency of the program. For example, if a data structure is accessed frequently but modified infrequently, then the program may be made more efficient by shifting the computation on the data structure from when it is accessed to when it is modified. Used to change the representation of a datatype, a collection of "views" can be integrated, or a high-level datatype definition can be implemented in a more concrete domain. For example, this transformation is an important step in implementing datatypes by components and is used in deriving an aggregate data structure for a text buffer in Section 3.1.5.

2.3.2 Shift

Using the *shift* transformation, computation is moved along the data paths of a program to increase the efficiency of the program (*eg.*, moving computation on a data structure from when it is accessed to when it is generated). This may change the data representation of a datatype but the meaning of the abstract datatype remains the same.

Example. Here is an example of the transformation of a data representation. The computation of the length of a sequence is shifted from access time to creation time by using the *shift* transformation. The datatype for the natural numbers contains the operations zero and

add-one. An additional operation, count, for converting a natural number into an integer is added.

Here, the Clu [54] language is used for the example. The abstract interface and implementation are defined together in a Clu cluster. The keyword cvt denotes the abstract datatype being defined. Operations defined on a type are referenced `type$op`, for example, the cons operation on lists is denoted list$cons.

```
Ndef = cluster is
    zero, add-one, count;
    rep = int List;
        zero = proc() returns (cvt);
            return(list$nil);
        end zero;
        add-one = proc(s:  cvt) returns (cvt);
            return(list$cons(1, s));
        end add-one;
        count = proc(s:  cvt) returns (int);
            return(list$length(s));
        end count;
```

The representation chosen is a sequence of 1's. Zero is represented by the empty sequence, adding one by concatenating 1 onto the sequence, and the count is obtained by taking the length. If count is accessed frequently, then the computation of counting the number of elements in the sequence may be shifted to creation time in the generator functions zero and add-one.

After several transformation steps (which have been omitted for the sake of brevity), the following is obtained:

```
Ndef = cluster is
    zero, add-one, count;
    rep = int;
        zero = proc() returns (cvt);
            return(0);
        end zero;
        add-one = proc(n:  cvt) returns (cvt);
            n := n + 1;
            return(n);
        end add-one;
        count = proc(n:  cvt) returns (int);
            return(n);
        end count;
```

In the new implementation, the computation is shifted away from count so that count simply looks up the value of the number directly. Thus, the transformation allows one to get from one representation to another in a controlled manner.

Description. The abstract interface of the datatype that follows has an operation that generates the datatype, and an operation that returns some information about the datatype.

Interface `Dtype` **is**
  gen : `Dtype`
  obs : `Dtype` $\rightarrow \beta$
end

The observer operation obs that returns some information about the datatype performs some computation, represented by its body O.

Repn `Dtype` **is** α
  with
    gen $\Leftarrow$ Abs(G)
    obs(Abs(a)) $\Leftarrow$ $O(a)$
  end

Using the *shift* transformation, the work is moved to the operation that generates the datatype, so that getting information about the datatype is now a simple lookup.

Repn `Dtype` **is** β
  with
    gen $\Leftarrow$ Abs'(O(G))
    obs(Abs'(a)) $\Leftarrow$ a
  end

Some of the expressiveness of the generators may be lost since it is filtered out by O; but this does not matter since other types can only access this type through the observer operations.

Using the Transformation. This transformation is used to optimize datatypes. It is frequently used after the other transformations that affect the abstract interfaces. Once operations are grouped into the desired context, then a *shift* is typically done to specialize the result. This transformation is used after the aggregate data structure is derived to specialize the buffer operations in this new context (see Section 3.1.6).

2.3.3 Expose

The *expose* transformation is a "synthetic" approach to revealing the underlying data structure of an existing datatype. This has the effect of moving the boundary of the type "inward." The data representation of the datatype changes but the meaning of the abstract datatype remains the same.

Example. Here is a datatype for valuations which is a table of mappings from the variables in a program to their values. This datatype has operations to create a new instance of a valuation, lookup a variable to obtain its value, and to enter a new variable-value mapping. Modula 3 [14] notation is used to represent this datatype. The abstract interface is specified by a Modula-3 interface definition.

```
INTERFACE Vln;
    TYPE T;
    PROCEDURE Nullvin() :  T;
    PROCEDURE Lookup(v :  T; i :  Id) :  Value;
    PROCEDURE Adjoin(v :  T; i :  Id; x :  Value) :  T;
END Vln.
```

The implementation is specified by a Modula-3 module definition. The initial representation is a collection of variable, value pairs.

```
MODULE Vln;
    TYPE T = LIST OF RECORD i :  Id; v :  Value END;
    PROCEDURE Nullvin() :  T =
        BEGIN RETURN nil END Nullvin;
    PROCEDURE Lookup(v :  T; i :  Id) :  Value =
        VAR p :  T;
        BEGIN
            p := assoc(v, i); RETURN ti(p)
        END Lookup;
    PROCEDURE Adjoin(v :  T; i :  Id; x :  Value) :  T =
        BEGIN RETURN cons(cons(i,x), v) END Adjoin;
BEGIN
END Vln.
```

An alternative way to implement this datatype is to store the value in memory with the variable associated with the memory "location."

```
MODULE Vln;
    TYPE Env = RECORD i :  Id; l :  Loc END;
    TYPE State = RECORD l :  Loc; v :  Value END;
    TYPE T = RECORD e :  LIST OF Env; s :  LIST OF State END;
    PROCEDURE Nullvin() :  T =
        VAR t :  T;
        BEGIN
            t.e := nil; t.s := nil; RETURN t
        END Nullvin;
    PROCEDURE Lookup(v :  T; i :  Id) :  Value =
        VAR p :  Env;
        BEGIN
            p := assoc(v.e, i); RETURN ti(assoc(v.s, ti(p)))
        END Lookup;
    PROCEDURE Adjoin(v :  T; i :  Id; x :  Value) :  T =
        VAR l :  Loc; t :  T;
        BEGIN
            l := genloc(); t.e := cons(cons(i, l), e); t.s := cons(cons(l,x), s); RETURN t
        END Adjoin;
BEGIN
END Vln.
```

This underlying location is "exposed" in the transformation process of the initial implementation to explicitly reveal the mapping of variables to locations, and locations to values. In an optimizing compiler, the mapping of variables to locations could be done at compile-time, while now only the look up of the value at the location would be done at runtime. The function genloc() generates a new location in memory.

One slight modification to Modula-3 is made to enhance the clarity of the example. A list constructor "LIST OF α" is introduced. Modula-3 does not have a list constructor, but this could be implemented using,

TYPE T == **REF RECORD** *d* : Data; *link* : T **END**;

Description. The abstract interface for a representative collection of operations on the datatype follows.

Interface Dtype **is**
gen : Dtype
ext : Dtype → Dtype
obs : Dtype → β
end

The operations are defined on the data representation α.

Repn Dtype **is** α
with
gen ⇐ Abs(G)
ext(a) ⇐ Abs(E(Rep(a)))
obs(a) ⇐ O(Rep(a))
end

Applying the *expose* transformation reveals the underlying data structure, the tuple $(\alpha_1 \times \ldots \times \alpha_n)$, and produces new implementations of the operations.

Repn Dtype **is** $(\alpha_1 \times \ldots \times \alpha_n)$
with
gen ⇐ $\langle Abs_1, \ldots, Abs_n \rangle (G')$
ext(a) ⇐ $\langle Abs_1, \ldots, Abs_n \rangle (E'(\langle Rep_1, \ldots, Rep_n \rangle (a)))$
obs(a) ⇐ $O'(\langle Rep_1, \ldots, Rep_n \rangle (a))$
end

The collection of abstraction functions, $\langle Abs_1, \ldots, Abs_n \rangle$, is applied to an n-tuple to create an n-tuple of abstractions. The collection of representation functions is similarly defined.

Using the Transformation. The *expose* transformation is useful for adapting a datatype to take advantage of special hardware or for revealing some information about the datatype that could be partially evaluated at compile-time. This transformation is used to decouple the buffer from the screen when introducing multiple windows to the editor example in Chapter 4.

2.3.4 Incorporate

The *incorporate* transformation is useful for specializing modules in the context they appear by moving external functions or subcomponents into a module. This changes the interface but does not interfere with the existing system because the transformation is only applied if dependencies (*eg.*, dataflow) within the program are preserved.

Example. Here is one step in the example that is shown in Section 2.3.6. A simple abstract datatype is used in a programming language interpreter for storing the bodies of subroutine definitions. The function MkFdef is used to create a subroutine definition given its body (an expression) and formal parameter names. The functions Body and Vars select the components. Ada [10] packages are used to specify the abstract interface.

```
Package FDEF is
    type Fdef is private;
        function MkFdef(E : in Exp, V : in VarList) return Fdef;
        function Body(F : in Fdef) return Exp;
        function Vars(F : in Fdef) return VarList;
    end FDEF;
function Fcode(F : in Fdef) return Codetype;
```

The external function Fcode creates the code necessary to execute the subroutine at runtime. It is incorporated into the datatype as a prelude to further specialization.

```
Package FDEF is
    type Fdef is private;
        function MkFdef(E : in Exp, V : in VarList) return Fdef;
        function Body(F : in Fdef) return Exp;
        function Vars(F : in Fdef) return VarList;
        function Fcode(F : in Fdef) return Codetype;
    end FDEF;
```

Description. External functions are made public functions of a type that has the same external scope. Likewise, imported modules are made substructures. There is a potential naming conflict with private functions (which would have to be renamed) but not with the public functions because they are in the same scope. Public functions of a type can be made private if there are no references to them in the external scope.

```
Interface Dtype is
    gen :  Dtype
    ext :  Dtype → Dtype
    obs :  Dtype → α
    end
fun :  β → γ
```

Applying the *incorporate* transformation includes the external function into the interface of the datatype.

```
Interface Dtype is
    gen :  Dtype
    ext :  Dtype → Dtype
    obs :  Dtype → α
    fun :  β → γ
    end
```

Using the Transformation. The *incorporate* transformation is often used as a prelude to specialization. Operations or modules could be grouped together for example, and then additional transformations applied to take advantage of the close coupling (*eg.*, to unfold calls to functions in the type). This transformation is used in Section 4.1.3 to group the buffer and the screen to get better performance.

2.3.5 Release

The *release* transformation is useful for removing unwanted code after a specialization step. It can be considered the opposite of incorporate because it moves functions or subcomponents outside of a module.

Example. Here is another step of the example that is shown in Section 2.3.6. After Fcode (which calls Body and Vars) is incorporated into the datatype, it is determined that there are no external references to Body and Vars. Ada [10] packages are used to specify the abstract interface.

```
Package FDEF is
    type Fdef is private;
        function MkFdef(E : in Exp, V : in VarList) return Fdef;
        function Body(F : in Fdef) return Exp;
        function Vars(F : in Fdef) return VarList;
        function Fcode(F : in Fdef) return Codetype;
    end FDEF;
```

The functions Body and Vars are released from the datatype since they are no longer used.

```
Package FDEF is
    type Fdef is private;
        function MkFdef(E : in Exp, V : in VarList) return Fdef;
        function Fcode(F : in Fdef) return Codetype;
    end FDEF;
```

Description. Private functions in a type are made public assuming no name conflicts are introduced into the external scope. Public functions that do not contain any instances of Rep or Abs for the defined type can be made external, provided that all private functions called by any of the public functions are made public. Likewise, substructures can be made a separate module and then imported.

Interface `Dtype` **is**
gen: `Dtype`
ext: `Dtype` → `Dtype`
obs: `Dtype` → α
fun: $\beta \rightarrow \gamma$
end

Applying the *release* transformation removes the function from the datatype.

Interface `Dtype` **is**
gen: `Dtype`
ext: `Dtype` → `Dtype`
obs: `Dtype` → α
end
fun: $\beta \rightarrow \gamma$

Using the Transformation. The *release* transformation is typically used as a prelude to specialization, or to remove operations that are no longer useful after a specialization. This transformation is used in Section 3.1.6 to remove redundant information from the buffer aggregate once final design decisions are made.

2.3.6 Strategies for Using the Module Transformation Rules

The transformations are typically used together, for example, to move boundaries as a prelude to specialization or shifting. Here is an example taken from Jørring and Scherlis [48], of one such transformation on a data representation that has a direct effect on program performance. A simple abstract datatype is used in a programming language interpreter for storing the bodies of subroutine definitions.

Interface `Fdef` **is**
MkFdef: `Exp` × `Var`* → `Fdef`
Body: `Fdef` → `Exp`
Vars: `Fdef` → `Var`*
end

The function MkFdef is used to create a subroutine definition given its body (an expression) and formal parameter names. The functions Body and Vars select the components. When a subroutine definition is encountered by the interpreter, MkFdef is called to create the corresponding object. `Fdefs` are represented as pairs.

```
Repn Fdef is Exp × Var*
    with
        MkFdef(E, V*)   ⇐   Abs(cons(E, V*))
        Body(D)   ⇐   hd(Rep(D))
        Vars(D)   ⇐   tl(Rep(D))
    end
```

Subroutine calls are carried out by the function Apply. When the subroutine F with the actual parameters V^* is encountered, the name of the subroutine is looked up in the program environment *PEnv* to obtain the `Fdef` consisting of the subroutine body and formal parameter names. The function Apply has two phases. It constructs the runtime code needed for executing the subroutine in the first phase. Then it applies the actual parameters to the code within the program environment in the second phase.

```
Apply(F, V*, PEnv)   ⇐   let D = find(PEnv, F) in
                             let code = Phase1(Body(D), MkEnv(Vars(D))) in
                                 Phase2(code, V*, PEnv)
```

Observation: The first phase could be carried out less frequently if it were done at the time that subroutines are defined rather than when they are called. This is accomplished by: *abstracting* Phase1 into the function Fcode, *incorporating* Fcode into the type signature, *shifting* Phase1 from Fcode to MkFdef, and *releasing* Body and Vars from the datatype since they are no longer referenced.

```
Repn Fdef is Codetype
    with
        MkFdef(E, V*)   ⇐   Abs'[Phase1(E, MkEnv(V*))]
        Fcode(D)   ⇐   Rep'[D]
    end

Apply(F, V*, PEnv)   ⇐   let D = find(PEnv, F) in
                             let code = Fcode(D) in
                                 Phase2(code, V*, PEnv)
```

In the new implementation, two things are happening: context outside of the type is incorporated into the type, that is, the boundary is widened; and computation is shifted from when subroutines are called to when they are defined, that is, internal data refinement. The transformation allows the software developer to get from one representation to another in a controlled manner.

2.4 Strategies in Constructing Systems Using this Approach

The three methods of integrating components, adding new components, and adapting module interfaces described in Section 2.1 are part of a similar integration process. The end result of each process is a collection of data transform procedures. The module transformation rules described in Section 2.3 can then be applied to them to yield efficient implementations. Putting the rules and the integration processes together, a process for constructing systems using this approach is defined.

The essential steps for implementing datatypes or modules by components:

1. Specify the interface. The names of the operations are listed with their signature.
2. Define the component implementations, each of which implements some subset of the interface. Collectively, all of the components implement the entire interface.
3. Establish any data invariances among the components by defining functions that translate from one component into another to establish the consistency of the collection of data representations.
4. Choose as an "expedient" representation the product of the component representations.
5. Integrate the components to define the datatype or module. Each operation defined in a component induces a corresponding operation on the aggregate datatype or module. Each operation definition is put into a format amenable to transformation.
6. Implement the datatype or module. Since the definitions are data transform procedures, this is done by applying transformations. The expression procedures are transformed into functional definitions. When adapting the system, it may be possible to reuse some of the information from the derivation of the integration of the original system.
7. Uncover an efficient representation by eliminating unnecessary redundancy and specializing data in the context that it appears in.
8. Implement an efficient implementation by translation.

Figure 2.1 (seen at the beginning of this chapter) is an abstract view of this process. The process starts with the *design* of the top-level aggregate specification (steps 1, 2 and 3). The software designer who wishes to design a complex system is able to decompose the problem into components that best model that portion of the problem. The components may be obtained from a library or prototyped by the designer using a data representation that most closely models the subproblem. Consistency relations establish correspondences among the data representations. This definition is used in the *integration* phase to produce an aggregate definition (steps 4 and 5). Obtaining the aggregate definition from the aggregate specification is a mechanical process that can be expressed as an algorithm (see Section 6.1.4) once compatibility maps (which respect the consistency relations) are provided. The aggregate definition is in a format upon which data translations (Section 2.3.1) can be performed to obtain an executable *prototype* (step 6). Then additional transformations such as *expose*, *incorporate*, *release*, and *shift* (Section 2.3) are performed to optimize the prototype into an efficient *implementation* (steps 7 and 8). Later on the software designer may wish to introduce additional functionality in the *adapt* phase.

Chapter 3

Integrating Module Interfaces: Deriving and Manipulating an Edit Buffer

In order to demonstrate the techniques for integrating module interfaces by program-transformations, we now go through an exercise in the development of a simple interactive text editor. We work through the derivation of the text-buffer implementation in Section 3.1. Then we introduce additional functionality in Section 3.2 to examine how the text buffer is adapted. Important concepts (such a "component") are informally introduced here, with their name in *italics*. A glossary of these terms is available in Appendix B. All such names in italics are precisely defined in Chapter 6.

Datatype definitions are represented as modules written in a notation based on an extended form of Standard ML [60]. The extensions add notation that is described as it is introduced in the examples. Although the decision was made to base the notation on Standard ML, as shown in the previous chapter, the transformation techniques are language independent and can be applied to other languages with modules such as Ada, Clu, and Modula 3. Standard ML also has some high-level abstraction mechanisms to facilitate the design of datatypes and that allow one to focus on the module structure and overall architecture of the system rather than getting lost in the details (which, although important at later stages in design and development, obscure the system structure that one is focusing on).

Other notations to consider are specification languages that support the design of large programs such as Larch [38] and Z [84]. Indeed, the style of developing systems in these languages has influenced the style of the editor-derivation presentation. However, since specifications deal with a higher level of abstraction, it is necessary to use a language in which abstract interfaces and data representations can be defined and manipulated. The decision was made to use Standard ML because it has an elegant module facility and fully defined semantics [61]. Moreover, Extended ML adds a useful extension to the language, the ability to add axioms. This gives the software designer a wide-spectrum language to represent higher-level specifications that can be refined to an implementable subset. See, for example, Sannella and Tarlecki [72], where a methodology for software development is developed using ML modules extended with axioms.

3.1 Deriving the Buffer

In this derivation, the editor is designed initially as a collection of separate components. The collection of components is integrated by deriving new module interfaces, resulting in an executable prototype. Then efficiency transformations are applied. The entire process is depicted in Figure 3.1, which is referred to in the example as the steps are elaborated.

3.1.1 Program Design

We are now ready to design the datatype of the text buffer. We start by defining the abstract interface of the datatype. An *abstract interface* is simply a signature. We use a Standard ML-like `signature` declaration to specify abstract interfaces.

The following specification defines an abstract interface for the datatype `buf` and the seven buffer operations: makebuf, delete, insert, move-left, move-right, show-char, and next-line.

```
Signature BUF = sig
  type buf
    makebuf :  buf
    delete :  buf → buf
    insert :  ch × buf → buf
    move-left :  buf → buf
    move-right :  buf → buf
    show-char :  buf → ch
    next-line :  buf → buf
  end
```

The next step is to design the data structure of the text buffer. Our goal is to arrive at a single data representation that supports the efficient implementation of all of the operations. Since designing a data representation that is satisfactory for all of the operations may be difficult, we begin by implementing subsets of the operations — each subset comprising a *component* — and then try to integrate them later.

3.1.2 Program Composition

The move operations are conveniently implemented by using a representation that is a sequence of characters with an explicit index for the point where editing takes place. The point of editing is moved left by decrementing the index and moved right by incrementing the index. The character at the point of editing is retrieved by looking up the character in the text to the left of the index. (The n^{th} element of a sequence s is denoted $s[n]$.) The component implementation shown below is based on this representation. The notation used for the component definition is similar to the Standard ML `structure` declaration. It is called a `component` because, unlike a structure, not all of the operations in the signature need to be implemented. The declaration implements the operations in terms of the domain of integers and character sequences (_* being the sequence type constructor).

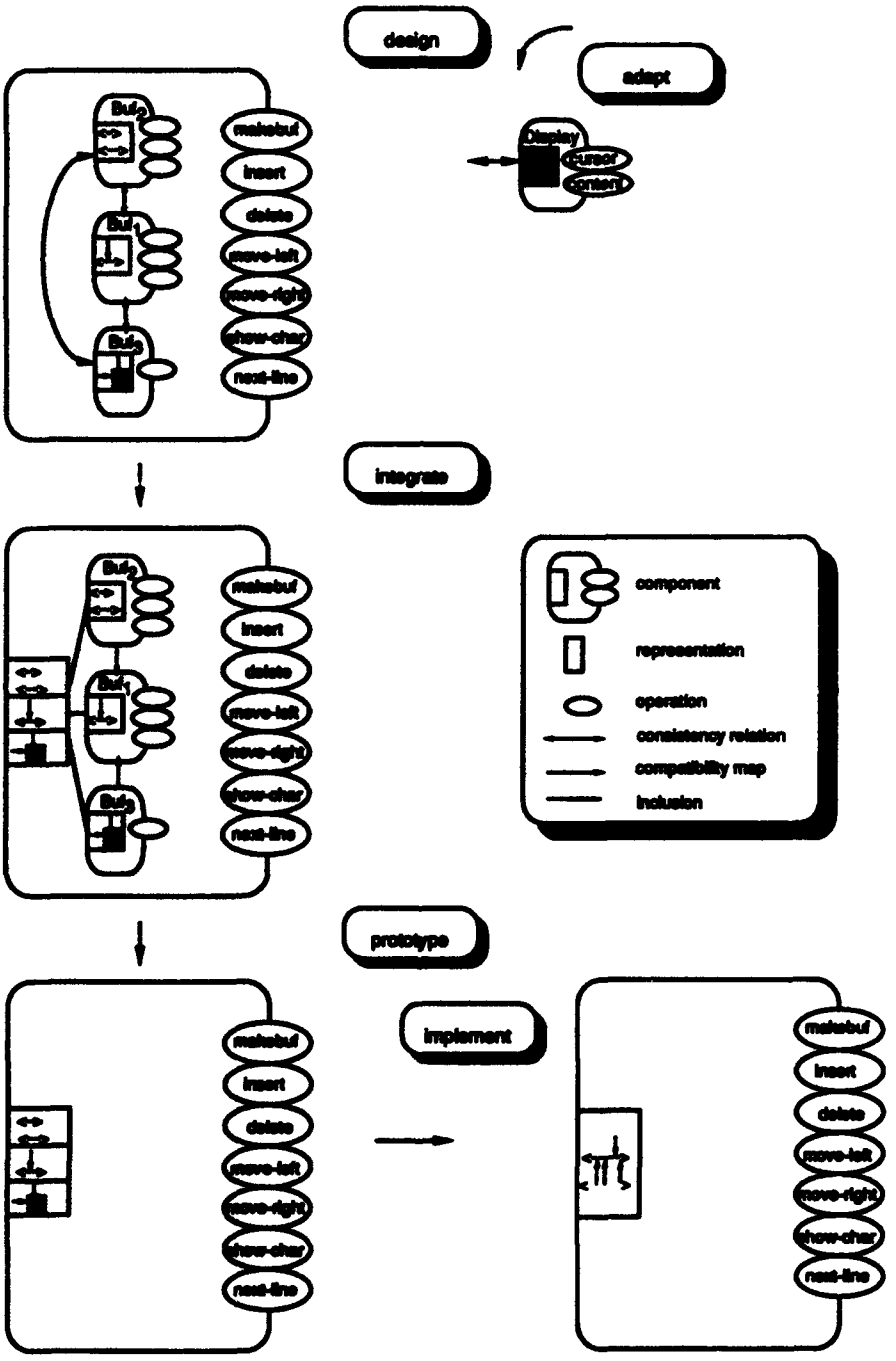


Figure 3.1: Deriving a Buffer

```
Component Buf1 : BUF = struct
  type buf = Buf of (int × ch*)
    move-left(Buf(p,t))   ⇐   Buf(p − 1, t)
    move-right(Buf(p,t))  ⇐   Buf(p + 1, t)
    show-char(Buf(p,t))   ⇐   t[p − 1]
    constraint Buf(p,t)   ⟹   0 ≤ p ≤ #t
  end
```

Abstraction of the underlying datatype representation is maintained using the datatype constructor `Buf`. Used on the lefthand side of the operation definition, the text-buffer representation is "revealed." Only operations defined within the datatype can use the datatype constructor in this manner. Used on the righthand side of the operation definition, the representation is "hidden." This provides an abstraction boundary where operations defined outside of the component are not allowed access to the data representation.

Constraints are axioms that contain additional information such as invariants of a datatype expressed in first order logic (where free variables are universally quantified and $\Longrightarrow$ is logical implication). The constraint

$$\texttt{Buf}(p,t) \quad \Longrightarrow \quad 0 \leq p \leq \#t$$

is an abbreviation for

$$\forall b : \texttt{buf}, \forall p : \texttt{int}, \forall t : \texttt{ch}^* \mid b = \texttt{Buf}(p,t) \,.\, 0 \leq p \leq \#t.$$

Extended ML [72] gives the developer this ability to add axioms to structures (and signatures). Constraints may be used as enabling conditions to provide additional context for transformations (see Appendix C). Constraints may also be refined in the implementation so that they are satisfied by each operation (see Section 3.1.6).

In the example, the axiom constrains the index to remain within the boundaries of the text ($\#s$ denotes the cardinality of the sequence s). The result of moving the index beyond the boundary is undefined at this stage in the design. Later on, as the component is refined into an implementation, more commitments about error handling can be made to ensure that each operation cannot violate the axiom. For example, the move operations could return the buffer unchanged if an attempt is made to move out of the buffer boundary. An alternative is to return an error value or flag so that a "beep" from the terminal could be emitted or the screen flashed. This would necessitate adapting the abstract interface which is discussed in the following chapter.

This component definition provides simple and natural definitions for the three operations shown. Implementing insertion and deletion of characters in this $\texttt{Buf}_1$ representation, on the other hand, requires an inconvenient (and hence error-prone) manipulation of subsequences within the text. A more convenient representation for these new operations is a pair of character sequences, representing the characters to the left and to the right of the point of editing. The index for the point of editing is left implicit. A character is deleted by removing the last element from the left sequence. A character is inserted by appending it to the left sequence, $l \bullet [c]$. There is no constraint on this component, though delete is unspecified when at the beginning of the buffer.

```
Component Buf2 : BUF = struct
    type buf = Buf of (ch* × ch*)
        makebuf   ⇐  Buf([ ], [ ])
        delete(Buf(l @ [c], r))   ⇐  Buf(l, r)
        insert(c, Buf(l, r))   ⇐  Buf(l @ [c], r)
    end
```

Although the makebuf operation is included in this component, it could have just as easily been defined in the first component as makebuf ⇐ Buf(0, [ ]).

The next-line operation moves the point of editing to the following line with the character position in the line remaining the same. This is difficult to implement using either of the previous representations, since it requires searching for newlines and computing the distance between the point of editing and the preceding newline. For this operation, then, a new component, $\texttt{Buf}_3$, is introduced where the text is a sequence of lines (where a line is a sequence of characters not containing a newline) and the point of editing is a line and character position. Then the point of editing is moved to the next line simply by incrementing the line position by one.

```
Component Buf3 : BUF = struct
    type line = (ch − 'nl')*
    type buf = Buf of ((int × int) × line*)
        next-line(Buf(⟨lp, cp⟩, ts))   ⇐  Buf(⟨lp + 1, cp⟩, ts)
        constraint Buf(⟨lp, cp⟩, ts)   ⇒  0 ≤ lp < #ts
        constraint Buf(⟨lp, cp⟩, ts)   ⇒  0 ≤ cp ≤ #ts[lp]
    end
```

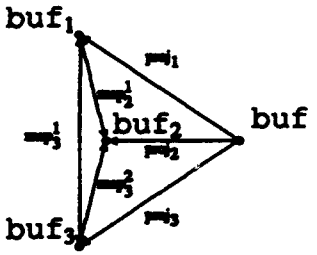
The newlines are implicit, giving a more compact representation. Of course, if one so chooses, an alternative representation can be used that keeps a newline character at the end of each line. The first invariant constrains the line pointer to remain within the boundaries of the buffer. The second invariant constrains the character index to remain within the line that it is on. As in the definition for $\texttt{Buf}_1$, the result of moving the point of editing beyond the boundary is undefined at this stage in the design. Not only does this happen when trying to move past the last character of a line, or past the last line of the buffer, but also when trying to use next-line to move from one line to the next that is shorter. This invariant can be refined at a later stage in the software development, for example, by moving to the same character position within the line if possible, but moving to the end of the line if the following line is shorter.

3.1.3 Aggregate Design

Collectively, these three components implement all of the operations of the abstract interface for BUF. However, in order to use all of the operations interchangeably, an "agreement" among the various data representations must be reached. These data representations are essentially "views" [29] or projections of some aggregate buffer. One way to reach agreement is to define *consistency relations* among the components, of the form $i \ \mathrm{map}_i^j \ j$. An

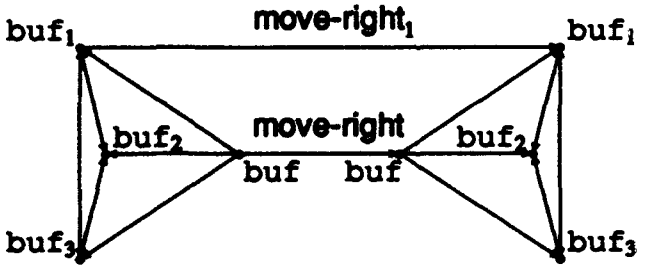
aggregate specification specifies a datatype constructed from a collection of components and consistency relations, and has the following properties: there are projection functions, which map the aggregate data object to an object of a particular component; and every component operation induces a corresponding operation in the aggregate that maintains the consistency of the components. The aggregate specification encapsulates the complexities that were avoided by not defining the operations in all of the components.

Using this definition in our example, then, an aggregate text-buffer is defined in terms of the components $\texttt{Buf}_1$, $\texttt{Buf}_2$, and $\texttt{Buf}_3$, along with the consistency relations that specify the agreement among these components (Figure 3.1, **design** step). A projection of an aggregate buffer data object yields an object of a particular component, and each such component object is related to other component objects by consistency relations (which is formally defined in Section 6.1.2). Intuitively, these relations provide a notion of consistency for the aggregate data objects.



In order to avoid ambiguity, names of types and operations are prepended with the name of the component in which it was defined. For example, each of the components define a type `buf` so $\texttt{Buf}_1$`.buf` refers to the type defined in the first component. To enhance the readability of the examples, these "qualified" names are abbreviated by omitting the component name, and using the component number as a subscript with the type or operation name. For example, $\texttt{buf}_1$ and $\textbf{move-right}_1$ are abbreviations for $\texttt{Buf}_1$`.buf` and $\texttt{Buf}_1$`.move-right`.

The effect of an operation on a text buffer is defined in terms of the corresponding component operation. In the aggregate, the operation must ensure that the projections of the buffer remain consistent. This is expressed in the following commutative diagram.



The relationships depicted in this diagram can be specified via axioms on the operations in the various components. Such axioms are written along with the abstract interface of the datatype (*eg.*, adding axioms to the abstract interface for the buffer shown in Section 3.1.1). Doing so results in an annotated abstract interface specification that includes a specification of component integration. As an example, the axioms for move-right are shown in Figure 3.2.

$$\textbf{axiom}\ \mathsf{proj}_1(\text{move-right}(b)) = \text{move-right}_1(\mathsf{proj}_1(b))$$

$$\begin{array}{lllll}
\textbf{axiom}\ \mathsf{proj}_1(b) & \mathsf{map}_1^2\ \mathsf{proj}_2(b) & \Longrightarrow & \mathsf{proj}_1(\text{move-right}(b)) & \mathsf{map}_1^2\ \mathsf{proj}_2(\text{move-right}(b)) \\
\textbf{axiom}\ \mathsf{proj}_2(b) & \mathsf{map}_2^3\ \mathsf{proj}_3(b) & \Longrightarrow & \mathsf{proj}_2(\text{move-right}(b)) & \mathsf{map}_2^3\ \mathsf{proj}_3(\text{move-right}(b)) \\
\textbf{axiom}\ \mathsf{proj}_3(b) & \mathsf{map}_3^1\ \mathsf{proj}_1(b) & \Longrightarrow & \mathsf{proj}_3(\text{move-right}(b)) & \mathsf{map}_3^1\ \mathsf{proj}_1(\text{move-right}(b))
\end{array}$$

Figure 3.2: Buffer Aggregate Specification for Move-Right

3.1.4 Aggregate Integration

An *aggregate definition* is a datatype that refines the aggregate specification. The data representation is defined as the product of the component data representations and the operations are defined in terms of the component operations as "data transform procedures" (Figure 3.1, integrate step).

Data transform procedures define alternative implementations on data representations in a way similar to the data transformation definitions presented in Section 2.2. They may take one of two forms:

1. Given a program f using a data representation D and a function, span, that translates elements of the data representation D to elements of the data representation D', we define f′ as:

$$\underline{f'}(\text{span}(d)) \Leftarrow \text{span}(f(d))$$

2. If, instead, there is a function, unspan, that translates elements of the data representation D' to elements of the data representation D, we define f′ as:

$$\text{unspan}(\underline{f'}(d)) \Leftarrow f(\text{unspan}(d))$$

This style of procedure definition is called an "expression procedure" by Scherlis [74]. While the expression procedure definition for f′ may not suggest an implementation, syntactic transformations can be applied to obtain a functional definition for the program f′ on the data representation D'. We saw examples of transforming data representations in Chapter 2 in the sections on the *translate*, *shift*, and *expose* transformations. We will see in Chapter 6 how they are explained in terms of data transform procedures. Since more than one operation may occur on the lefthand side of an expression procedure definition, there may be some confusion about which operation is being defined. Because of this, the operation being defined is underlined to distinguish it from the others.

Following this approach, a preliminary definition of move-right on the aggregate is obtained (Figure 3.3). Recall that move-right is defined in component $\texttt{Buf}_1$. Alternative versions of move-right for the other components (*i.e.*, move-right_2 and move-right_3) are defined as data transform procedures in terms of the $\texttt{Buf}_1$ definition once "compatibility maps" (*i.e.*, $\mathsf{map}_{i\to j}$ which serves as unspan) between the components have been defined.

```
map2→1(move-right2(b))  ⇐  move-right1(map2→1(b))
map3→1(move-right3(b))  ⇐  move-right1(map3→1(b))

move-right(Buf(p,t,l,r,⟨lp,cp⟩,ts))  ⇐
    Buf(p', t', l', r', ⟨lp',cp'⟩, ts')
        where Buf1(p',t')  =  move-right1(Buf1(p, t))
          and Buf2(l',r')  =  move-right2(Buf2(l, r))
          and Buf3(⟨lp',cp'⟩,ts')  =  move-right3(Buf3(⟨lp,cp⟩, ts))
```

Figure 3.3: Preliminary Definition of Move-Right

A *compatibility map* is a function that respects the consistency relation. It translates one component representation into another representation. Depending on the consistency relation, it may not be possible to implement compatibility maps in both directions, but normally it will be straightforward to implement one of them.

The alternative versions of move-right are defined by data transform procedures where the compatibility maps serve as the span or unspan functions. The data representation of the aggregate is the product of the component representations. The move-right operation on the data aggregate is defined as the *product* of the component operations, where each component operation updates the appropriate fields of the aggregate representation.

This definition is easy to construct, but is constrained, however. The implementations of move-right for the other components, move-right_2 and move-right_3, only make use of their "own" representations, that is, $\texttt{Buf}_2$ and $\texttt{Buf}_3$, respectively. Thus we are not able to take full advantage of any interrelationships among the various representations when deriving an implementation.

A more general approach is to arrange for all of the component representations to be available for the operation definitions in order to take advantage of any interrelationships among the various representations. Going back to the preliminary definition of the aggregate, instead of actually deriving implementations for the component operations, the data transform procedure itself can be symbolically manipulated to yield a new definition for the aggregate.

The buffer definition shown in Figure 3.4 uses this approach; here, the buffer operations (after the keyword in) are again defined by data transform procedures. In taking advantage of this added generality, the operations are defined in terms of the aggregate buffer representation, or some portion of this representation. Then implementations of the operations are derived with all component representations available. To map between the components and the various aggregates, "span" and "unspan" functions are used (after the keyword local in the figure)—this also has the effect of putting things into a form suitable for data transformation. The constraints on the aggregate (at the bottom of the figure) are obtained from the constraints on the components.

The translation functions, span and unspan, for the data transform procedures are defined in terms of the compatibility maps. A *span* function is a mapping from one component into the aggregate of all *reachable* components (*i.e.*, connected by compatibility

```
Structure Buf : BUF = struct
    structure Buf1, Buf2, Buf3
    type buf = Buf of (int × ch* × ch* × ch* × (int × int) × line*)
        local
            map2→1(Buf2(l,r))   ⇐   Buf1(#l, l @ r)
            map3→1(Buf3(⟨lp,cp⟩,ts))   ⇐
                Buf1(#(lines-to-chars(ts[..(lp − 1)])) + cp, lines-to-chars(ts))
                    where lines-to-chars(s) = if null(s) then [ ]
                                              else [hd(s)] @ ['nl'] @ lines-to-chars(tl(s))
            spana(Buf2(l,r))   ⇐
                Buf1×2(p, t, l, r)
                    where Buf1(p,t)   =   map2→1(Buf2(l, r))
            unspanb(Buf(p,t,l,r,⟨lp,cp⟩,ts))   ⇐
                Buf1×2({ p | p3 }, { t | t3 }, l, r)
                    where Buf1(p3,t3)   =   map3→1(Buf3(⟨lp,cp⟩, ts))
            unspanc(Buf(p,t,l,r,⟨lp,cp⟩,ts))   ⇐
                Buf1({ p | p2 | p3 }, { t | t2 | t3 })
                    where Buf1(p2,t2)   =   map2→1(Buf2(l, r))
                      and Buf1(p3,t3)   =   map3→1(Buf3(⟨lp,cp⟩, ts))
            ...
        in
            makebuf1×2   ⇐   spana(makebuf2)
            unspanb(makebuf)   ⇐   makebuf1×2
            ...
            unspanc(move-right(b))   ⇐   move-right1(unspanc(b))
            show-char(b)   ⇐   show-char1(unspanc(b))
            next-line1×3(spana(b))   ⇐   spana(next-line3(b))
            unspanc(next-line(b))   ⇐   next-line1×3(unspanc(b))
        end
        constraint Buf(p,t,l,r,⟨lp,cp⟩,ts)   ⟹   0 ≤ p ≤ #t
        constraint Buf(p,t,l,r,⟨lp,cp⟩,ts)   ⟹   0 ≤ lp < #ts
        constraint Buf(p,t,l,r,⟨lp,cp⟩,ts)   ⟹   0 ≤ cp ≤ #ts[lp]
    end
```

Figure 3.4: Buffer Definition

maps). An *unspan* function is a mapping from some aggregate of components into the component that is reached by all of them.

The definitions of the operations and spanning functions are not *ad hoc*: they are obtained mechanically by considering the order in which the components must be "merged." The basic idea is to consider the buffer definition as a graph, where the components are nodes and the compatibility maps are directed arcs. The operations for each component must be reimplemented to operate on the aggregate. This is done in a series of stages. Starting at the node representing the component where the operations are defined, all connected nodes are merged into a new "coalesced" node using a variant of the data transformation techniques. This "coalescing" of connected nodes is repeated until the graph collapses into a single node. (See Section 6.1.4 for more details.)

In the definition shown in Figure 3.4, `Buf` implements the abstract interface `BUF` using the components $\texttt{Buf}_1$, $\texttt{Buf}_2$, and $\texttt{Buf}_3$. A representative sample of operations is shown. Defining the makebuf operation for the aggregate requires two stages because the $\texttt{Buf}_2$ component, in which it is defined, is not directly connected to all the other components. It is connected directly to $\texttt{Buf}_1$ via a compatibility map, but indirectly to $\texttt{Buf}_3$. In the first stage, an intermediate definition of makebuf is defined, $\text{makebuf}_{1\times2}$, on an intermediate aggregate, $\texttt{Buf}_{1\times2}$, (the representation is the product of the $\texttt{Buf}_1$ and $\texttt{Buf}_2$ representations). In the second stage, the final operation on the aggregate buffer is defined by merging this intermediate definition with the $\texttt{Buf}_3$ component. Since the $\texttt{Buf}_1$ component is directly connected to all other components, new implementations for the operations defined in this component (*eg.*, move-right and show-char) are defined in a single step.

The components are kept consistent through the compatibility maps $\text{map}_{2\to1}$ and $\text{map}_{3\to1}$. It is not necessary that all translations among components be given; it is sufficient that the components are connected, possibly through some number of intermediate components. Component $\texttt{Buf}_2$ is mapped into component $\texttt{Buf}_1$ by making the point of editing explicit (which is the number of characters to the left of the point, $\sharp l$), and by appending the left and right sequence of characters together. Component $\texttt{Buf}_3$ is mapped into component $\texttt{Buf}_1$ by converting the line and character indices into a character index and by converting the sequence of lines into a sequence of characters. The auxiliary function lines-to-chars takes a sequence of lines, adds a newline to the end of each one and appends them to make a sequence of characters. (The notation $s[\_..i]$ denotes the subsequence of s from the beginning of s to i inclusive.) The translation functions are easily defined in terms of the compatibility maps. In the definitions of unspan_b and unspan_c, a value that is computed in more than one way is denoted $\{ v_1 \mid \ldots \mid v_n \}$, where v_i represents the value derived from component i. Multiple ways to compute a value are maintained to ensure consistency among the components.

3.1.5 Aggregate Prototype

Next, a prototype (Figure 3.5) is derived (Figure 3.1, prototype step) where the expression procedures defining the buffer operations are transformed into functional definitions. This results in an *aggregate prototype*, which is a refinement of the aggregate definition. For

```
Structure Buf_proto : BUF = struct
  type buf = Buf of (int × ch* × ch* × ch* × (int × int) × line*)
    makebuf  ⇐  Buf(0, [ ], [ ], [ ], ⟨0,0⟩, [ ])
    ...
    move-right(Buf(p,t,l,r,⟨lp,cp⟩,ts))  ⇐
      let lp',cp' = if (cp = #ts[lp]) then lp + 1, 0 else lp, cp + 1 in
        Buf(p + 1, t, l @ [hd(r)], tl(r), ⟨lp',cp'⟩, ts)
    show-char(Buf(p,t,l,r,⟨lp,cp⟩,ts))  ⇐
      { t[p − 1] | last(l) | (if (cp = 0) then 'nl' else ts[lp][cp − 1]) }
    next-line(Buf(p,t,l,r,⟨lp,cp⟩,ts))  ⇐
      let d = (nlpos(ts))[lp] − (nlpos(ts))[lp − 1] in
        Buf(p + d, t, l @ r[..d], r[(d + 1)..], ⟨lp + 1, cp⟩, ts)
    constraint Buf(p,t,l,r,⟨lp,cp⟩,ts)  ⇒  0 ≤ p ≤ #t
    constraint Buf(p,t,l,r,⟨lp,cp⟩,ts)  ⇒  0 ≤ lp < #ts
    constraint Buf(p,t,l,r,⟨lp,cp⟩,ts)  ⇒  0 ≤ cp ≤ #ts[lp]
  end
```

Figure 3.5: Buffer Prototype

brevity, the steps have been omitted. (The steps for transforming move-right are shown in Appendix C.) As with other data transformations, they consist of a number of purely mechanical steps and a few insight steps that require input from the designer. It is not actually necessary to derive translation functions that are computable. Instead, the transformation process makes use of them in syntactic manipulations to obtain computable functions for the buffer operations.

Typically, the transformation proceeds where: (1) the bodies of the old operation and span function are expanded; (2) domain knowledge about the data representation is applied to simplify the body; (3) an insight step is applied to "bridge" the old and new representations; (4) additional simplification steps are applied; and (5) the span function is abstracted from the body of the operation. The insights required by the user are knowledge of the properties of the domain and the underlying semantic model, and knowledge of using traditional transformations. For example, transforming move-right requires knowledge about the buffer domain that adding a character to the buffer changes the point of editing, and knowledge about the underlying model of sequences that a sequence is equivalent to the list formed by the first element appended to the rest of the sequence. The traditional transformation of case analysis is used to capture the constraints on the buffer domain.

In the prototype, the data representation is simply the product of the data representations of the components. All components are updated simultaneously. The makebuf operation generates each component representation. The move-right operation increments the index appropriately for each component (the functions hd and tl return the first element and the rest of a sequence). The show-char operation returns the character at the point of editing in the buffer (the function last returns the last element of a sequence). The value may be produced

```
Structure Buf_impl : BUF = struct
  type buf = Buf of ((int × ch*) × (int × int*))
    makebuf  ⇐  Buf(0, [ ], 0, [ ])
    ...
    move-right(b as Buf(p,t,i,nl))  ⇐
      if p ≥ #t then b
        else Buf(p + 1, t, (if nlp(t[p]) then i + 1 else i), nl)
    show-char(Buf(p,t,i,nl))  ⇐
      if p = 0 then ⟨sp⟩
        else t[p − 1]
    next-line(b as Buf(p,t,i,nl))  ⇐
      if (p + (nl[i] − nl[i − 1])) > #t then b
        else Buf(p + (nl[i] − nl[i − 1]), t, i + 1, nl)
  end
```

Figure 3.6: Buffer Implementation

from any of the three representations; these three alternatives are denoted $\{ v_1 \mid v_2 \mid v_3 \}$, where v_1 is $t[p-1]$ and so on. This extension could be easily implemented by selecting the first alternative so that the prototype could be executed. Multiple ways to compute a value are kept in order to avoid losing information that may be useful in later transformation or analysis steps. In the next-line operation, the positions of the surrounding newlines are used to advance to the next line for the $\texttt{Buf}_1$ component (the function nlpos takes a sequence of lines and returns a sequence of newline positions).

Notice that the character index for the $\texttt{Buf}_1$ component has been transformed to use information from the $\texttt{Buf}_3$ component, where it is easier to compute newline information. The representation of each component is available to update any other component representation. How they are used provides the motivation for the final representation that follows.

3.1.6 Aggregate Implementation

It is readily apparent that this prototype is not the most efficient implementation because of the redundancy in the data and the operations. The following observations are made: (1) It is not necessary to keep the three alternatives for show-char, so one of them is selected — as the developer, we choose $t[p-1]$, and make a commitment to using the $\texttt{Buf}_1$ component. (2) The data elements of the $\texttt{Buf}_2$ component, *l* and *r*, are not used in any operations to compute the data elements of the $\texttt{Buf}_1$ component, *t* or *p*, so they are removed. (3) The data elements of the $\texttt{Buf}_3$ component, *ts* and *lp*, are used to provide newline information for computing *p* in next-line. They are saved in this specialized context by *shifting* (via transformations) the time that the newline positions are computed from access to generation time. A new data representation is defined from this process of making commitments. This process can

be conceptualized as a mapping from the prototype representation to a new representation; the map function encapsulates the insights provided by the user such as constraints on the merged operations and the implementation (*eg.*, efficiency) considerations.

With these observations in mind, a specialized implementation (Figure 3.6) is derived using data transformation techniques to obtain a representation that caches newline positions (Figure 3.1, implement step). This *aggregate implementation* is a refinement of the prototype, providing an "efficient" implementation of the datatype. For brevity, the steps again have been omitted. Performance is improved by eliminating the computation for the $\texttt{Buf}_2$ component, and computing newline information directly rather than maintaining a sequence of lines and mapping it into newline positions when needed. The former is an instance of *releasing* components from the datatype, while the latter is an instance of *shifting* computation from access to generation time, techniques that are described in [48, 76]. A part of the derivation that illustrates shifting computation is shown in Appendix D.

The new specialized representation is a sequence of characters with an index for the point of editing and a sequence of newline positions with an index tracking which line contains the point of editing. The makebuf operation now generates an empty buffer and newline cache. The move-right operation updates the newline index when crossing over a line (the predicate nlp returns true when its argument is a newline). The next-line operation uses the newline cache to move more efficiently.

The constraints have been refined into the implementations of the operations by taking care of error handling for boundary conditions (to satisfy the constraints). They are satisfied in the move-right operation, for example, by returning the original buffer if an attempt is made to move the point of editing past the boundary.

3.2 Adapting the Buffer

At this point, the initial buffer has been successfully implemented. However, it often happens that users desire additional functionality. This section presents a mechanism for introducing change to the system (Figure 3.1, adapt step). Components for pages, regions, and s-expressions are introduced to adapt the core system consisting of the components $\texttt{Buf}_1$, $\texttt{Buf}_2$, and $\texttt{Buf}_3$ (see Figure 3.7). The display component, $\texttt{Buf}_{disp}$, is added in the following chapter. The component, $\texttt{Buf}_i$, serves as a useful intermediate for relating the component for s-expressions to the existing system. Compatibility maps are represented by arrows. For example, an arrow from $\texttt{Buf}_2$ to $\texttt{Buf}_1$ represents the compatibility map that takes an object of type $\texttt{Buf}_2$ and produces an object of type $\texttt{Buf}_1$. Although $\texttt{Buf}_{impl}$ is not shown in the figure, it does not mean that we must start all over again. We see in the examples that follow how the existing work is supplemented when adapting the buffer.

The components are chosen for their properties. The page component is similar to the original components and offers another view of the buffer. The region component introduces something new, the concept of a "mark." The s-expression component contains less information about the buffer than the original components, the compatibility map from the core to it is many-to-one. The goal of this exercise is to learn how the method handles adaptation and scaling, by adding components under a variety of conditions.

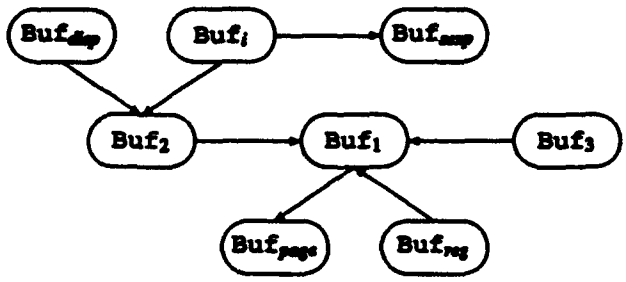


Figure 3.7: Buffer Components

3.2.1 Pages

The buffer is first extended to operate on pages in order to learn how the system is adapted by adding a similar kind of component, what alternatives are available, and the cost of adding a new component.

A page is a region of text delineated by a special page marker, typically the "control-L" character. We extend the buffer signature by adding the following operations:

backward-page : buf → buf
forward-page : buf → buf
what-page : buf → int

These operations allow movement to the previous or following page, and provide information on which page the point of editing is on. For these operations, we view the buffer as a sequence of pages with an index pointing to the current page.

Component $\mathtt{Buf}_p$: BUF = struct
  type page = (ch − '^L')*
  type buf = Buf of (int × page*)
    backward-page(Buf(*pi*, *tp*)) ⇐ Buf(*pi* − 1, *tp*)
    forward-page(Buf(*pi*, *tp*)) ⇐ Buf(*pi* + 1, *tp*)
    what-page(Buf(*pi*, *tp*)) ⇐ *pi*
  end

The page separators are implicit; of course, an alternative representation is possible that keeps the page separator at the end of each page. The choice is up to the designer.

Recall that the original system was defined using an annotated signature that specifies how to integrate the original three components (Figure 3.2). This is then enriched to include the component for pages by adding these axioms:

axiom proj_p(forward-page(*b*)) = forward-page$_p$(proj_p(*b*))

axiom proj_1(*b*) map_1^p proj_p(*b*) ⟹ proj_1(forward-page(*b*)) map_1^p proj_p(forward-page(*b*))
axiom proj_2(*b*) map_2^p proj_p(*b*) ⟹ proj_2(forward-page(*b*)) map_2^p proj_p(forward-page(*b*))
axiom proj_3(*b*) map_3^p proj_p(*b*) ⟹ proj_3(forward-page(*b*)) map_3^p proj_p(forward-page(*b*))

The first axiom specifies the behavior of the forward-page operation on the data aggregate in terms of the page component where it was defined. The remaining axioms ensure that the system remains consistent after the operation is performed.

Additionally, the axioms for the original operations must be extended to ensure consistency of the new component when an old operation is applied. For example, the axioms for move-right are extended with,

$$\textbf{axiom}\ \mathsf{proj}_{1}(b)\ \ \mathsf{map}_{1}^{p}\ \ \mathsf{proj}_{p}(b) \implies \mathsf{proj}_{1}(\text{move-right}(b))\ \ \mathsf{map}_{1}^{p}\ \ \mathsf{proj}_{p}(\text{move-right}(b)).$$

How do we integrate this component with our previously developed buffer? We consider two alternatives: (1) *Merging with the original system.* We could start over by integrating all of the components again, and developing a new prototype and implementation. We will see that much of the implementations for the operations of the existing components can be reused. (2) *Translating into the original system.* We could choose the representation of the original system and then we would only need to integrate the new component without modifying the operations of the existing system.

Merging with the Original System. In order to merge the new component with the original system, first we must define the relationship between the component and the original buffer, since the component is a new view of the buffer. This is accomplished by defining a compatibility map that translates between the new component and one of the previously defined components. Here we choose $\mathtt{Buf}_1$, where the buffer was represented as an index marking the point of editing, and a sequence of characters.

$$\mathsf{map}_{1\to p}(\mathtt{Buf}_1(p,t)) \Leftarrow \mathtt{Buf}_p(\mathsf{npages}(t[_p - 1]), \mathsf{chars2pages}(t))$$

We map $\mathtt{Buf}_1$ into $\mathtt{Buf}_p$ by counting the number of page markers preceding the text index (the auxiliary function, npages, returns the number of page markers in the text) to get the index for the page in which the point of editing occurs, and by parsing the text into a sequence of pages (using chars2pages). We must then go through the process of integrating the components to obtain a prototype. We can reuse the implementations for the operations of the existing components directly.

Additionally, we must derive a new implementation for each page operation in each of the existing components, $\mathtt{Buf}_1$, $\mathtt{Buf}_2$, and $\mathtt{Buf}_3$. We start by extending the definitions of Figure 3.4. For example, the definition for forward-page follows.

local

$$\mathsf{unspan}(\mathtt{Buf}_{1\times p}(p,t,pi,tp)) \Leftarrow \{\, \mathsf{map}_{1\to p}(\mathtt{Buf}_1(p,t)) \mid \mathtt{Buf}_p(pi,tp) \,\}$$

$$\mathsf{unspan}'(\mathtt{Buf}'(p,t,l,r,\langle lp,cp\rangle,ts,pi,tp)) \Leftarrow \mathtt{Buf}_{1\times p}(\{\, p \mid p_2 \mid p_3 \,\}, \{\, t \mid t_2 \mid t_3 \,\}, pi, tp)$$

where $\mathtt{Buf}_1(p_2,t_2) = \mathsf{map}_{2\to 1}(\mathtt{Buf}_2(l,r))$

and $\mathtt{Buf}_1(p_3,t_3) = \mathsf{map}_{3\to 1}(\mathtt{Buf}_3(\langle lp,cp\rangle, ts))$

in

$$\mathsf{unspan}(\underline{\text{forward-page}}_{1\times p}(\mathtt{Buf}_{1\times p}(p,t,pi,tp))) \Leftarrow \text{forward-page}_{p}(\mathsf{unspan}(\mathtt{Buf}_{1\times p}(p,t,pi,tp)))$$

$$\mathsf{unspan}'(\underline{\text{forward-page}}(\mathtt{Buf}'(p,t,l,r,\langle lp,cp\rangle,ts,pi,tp))) \Leftarrow \text{forward-page}_{1\times p}(\mathsf{unspan}'(\mathtt{Buf}'(p,t,l,r,\langle lp,cp\rangle,ts,pi,tp)))$$

end

The intermediate definition for forward-page$_{1\times p}$ (appearing after the keyword in) is novel, since it defines how the newly introduced component relates to an established one, namely, $\texttt{Buf}_1$. The definition for forward-page that follows it is similar to the previous definitions for the operations of $\texttt{Buf}_1$, but with the addition of the $\texttt{Buf}_p$ component. For example, compare this definition with the one in Figure 3.4.

We must also derive a new implementation for each operation of the existing components in the new page component. For example, the move-right operation (defined in Figure 3.4) must be extended to define the operation on the new aggregate that includes the new page component in terms of the old aggregate. This is done by simply adding a new definition.

```
local
    span(Buf(p,t,l,r,⟨lp,cp⟩,ts))  ⇐  Buf'(p, t, l, r, ⟨lp,cp⟩, ts, pi, tp)
                                          where Buf_p(pi,tp)  =  map_{1→p}(Buf_1(p, t))
in
    move-right'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
        span(move-right(Buf(p, t, l, r, ⟨lp,cp⟩, ts)))
end
```

We are able to reuse all of the old definitions of the core system directly. We are able to reuse parts of the derivation as well, having only to derive new implementations based on the above definitions. These steps are similar to those done in the core system, and yield an executable prototype. For the implementation step, we might choose to specialize the pages in a similar manner as we specialized lines, keeping a sequence of page positions and an index pointing to the current page.

Translating into the Original System. When translating the new component into the original system, the data structure and operations remain unchanged in the prototyping step. It is only necessary to derive a new implementation for each page operation into the original system. For example, the definition for forward-page follows.

```
local
    unspan(Buf_1(p,t))  ⇐  map_{1→p}(Buf_1(p, t))
    unspan'(Buf'(p,t,l,r,⟨lp,cp⟩,ts))  ⇐
        Buf_1({ p | p_2 | p_3 }, { t | t_2 | t_3 })
            where Buf_1(p_2,t_2)  =  map_{2→1}(Buf_2(l, r))
              and Buf_1(p_3,t_3)  =  map_{3→1}(Buf_3(⟨lp,cp⟩, ts))
in
    unspan(forward-page_1(Buf_1(p,t)))  ⇐  forward-page_p(unspan(Buf_1(p, t)))
    unspan'(forward-page(Buf'(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
        forward-page_1(unspan'(Buf'(p, t, l, r, ⟨lp,cp⟩, ts)))
end
```

Compare this definition for translating with the one given previously for merging. There are the same number of definitions, but the new page component is not included in the aggregate. Also notice that a new definition is not needed for any of the existing operations, such as move-right, because the aggregate representation does not change. For the implementation step, the aggregate can be specialized as before.

3.2.2 Regions

The example is now extended to provide operations that manipulate regions. Recall that the previous example introduced a new view of the text buffer which is computed via a compatibility map from one of the previous components. A region, however, introduces a new data construct, "mark," that cannot be so computed. This section identifies how this new data construct interacts with the other components.

A region is a portion of the text delineated by the point of editing and a special marker, called the *mark*.

```
set-mark:      buf → buf
exchange:      buf → buf
delete-region: buf → buf
```

The operations allow one to set the mark to the current value of point, exchange the values of mark and point, and to delete the text between mark and point. For these operations, we view the buffer as simply consisting of a point, a mark, and the text.

```
Component Buf_r : BUF = struct
    type buf = Buf of (int × int × ch*)
        set-mark(Buf(p,m,t))    ⇐  Buf(p, p, t)
        exchange(Buf(p,m,t))    ⇐  Buf(m, p, t)
        delete-region(Buf(p,m,t)) ⇐  let (i,j) = if p < m then p,m else m,p in
                                        Buf(i, i, t[..i − 1] @ t[j..])
    end
```

The initialization of the mark is handled by the makebuf operation as a result of the region component being integrated with the existing system.

A compatibility map between this component on regions and one of the existing ones is easy to define. We choose $\texttt{Buf}_1$ since its fields are a subset of $\texttt{Buf}_r$.

$$\text{map}_{r\to 1}(\texttt{Buf}_r(p,m,t)) \Leftarrow \texttt{Buf}_1(p, t)$$

The integration is simple. For example, the definition for set-mark follows.

```
local
    span(Buf_r(p,m,t)) ⇐
        Buf_1×r(p', t', p, m, t)
            where Buf_1(p',t')  =  map_r→1(Buf_r(p, m, t))
    unspan(Buf'(p,t,l,r,⟨lp,cp⟩,ts,p,m,t)) ⇐
        Buf_1×r({ p | p2 | p3 }, { t | t2 | t3 }, p, m, t)
            where Buf_1(p2,t2)  =  map_2→1(Buf_2(l,r))
              and Buf_1(p3,t3)  =  map_3→1(Buf_3(⟨lp,cp⟩, ts))
in
    set-mark_1×r(span(b))  ⇐  span(set-mark_r(b))
    unspan(set-mark(b))    ⇐  set-mark_1×r(unspan(b))
end
```

Since p and t are identical to the components of $\texttt{Buf}_1$, they can "share" the same data representation. The first definition then simplifies, since set-mark$_{1\times r}$ is identical to set-mark$_r$. This is a special case where (part of) the compatibility map is the identity function. The second definition also simplifies since the only change is in adding the mark. The additional data, m, is added without affecting any of the existing components since it is "orthogonal" to the other data elements, unlike the previous page component which was a different "view" of the buffer.

This also simplifies the changes to the existing operations. For example, the move-right operation can be extended to define the operation on the new aggregate that includes the new region component in terms of the old aggregate. This is done by simply adding a new definition. However, this definition simplifies to the original definition with an added field that is simply passed along.

local
  unspan(Buf($p,t,l,r,\langle lp,cp\rangle,ts,p,m,t$)) $\Leftarrow$
    Buf'({ p | p }, { t | t }, l, r, $\langle lp,cp\rangle$, ts)
      where $\texttt{Buf}_1(p,t)$ = $\text{map}_{r\to 1}(\texttt{Buf}_r(p, m, t))$
in
  unspan(move-right'(b)) $\Leftarrow$ move-right(unspan(b))
end

The point of this example is, there is a mechanism for introducing change without modifying the original system. Even if we only want to add an operation to an existing component, we introduce an additional component, though it may have an identical representation to an existing one. This preserves the integrity of the existing system and also records the adaptation. Transformation techniques allow one to optimize the aggregate by sharing representations.

3.2.3 S-Expressions

The example is now extended to provide operations to manipulate s-expressions, nested parenthesized expressions. This is an interesting example of adapting the data aggregate because the compatibility map between the aggregate and the new component is not one-to-one, since information about white space is lost when parsing from text to s-expressions. Here an operation for moving over a single s-expression is shown.

move-sexp: buf $\rightarrow$ buf

The representation of the component, $\texttt{Buf}_s$, consists of a pair of sequences of s-expressions, to the left and right of the point of editing. Furthermore the left sequence is reversed so that both sequences can be reasoned about identically (*e.g.*, as stacks).

Component $\texttt{Buf}_s$: BUF = **struct**
  type buf = Buf **of** (sexp* $\times$ sexp*)
    move-sexp(Buf(ls,rs)) $\Leftarrow$ Buf(tl(ls), [hd(ls)] @ rs)
  end

The semantics of s-expressions is sensitive to the context in which the point of editing appears. An alternative semantics could be to parse from the beginning of the text. However, when inside a nested s-expression, move-sexp is only applicable to s-expressions at that level of nesting, so information about s-expressions in the enclosing nesting levels is not needed. If operations to ascend and descend nesting levels were to be defined, then a different representation for the type would be required.

This component is linked to the core system through an intermediate component $\texttt{Buf}_i$ (for convenience) that has no operations. The component represents the buffer as a pair of sequences of characters to the left and right of the point of editing. Like $\texttt{Buf}_s$, the left sequence is is reversed so that both sequences can be reasoned about identically.

```
Component Buf_i : BUF = struct
    type buf = Buf of (ch* × ch*)
    end
```

The text is parsed into a sequence of s-expressions, starting from the point of editing, and proceeding in both directions.

```
local
    parse  ⇐  fn [].[]
                | cons(a,x).if a ∈ ['a'..'z','0'..'9'] then cons(⟨sexp⟩, parse(parse-an(x)))
                                  else if a = '(' then cons(⟨sexp⟩, parse(parse-sexp(x)))
                                      else if a = ')' then []
                                          else parse(x)

    parse-an  ⇐  fn [].[]
                   | cons(a,x).if a ∈ ['a'..'z','0'..'9'] then parse-an(x) else cons(a, x)

    parse-sexp  ⇐  fn [].error
                     | cons(a,x).if a = ')' then x
                                  else if a = '(' then parse-sexp(parse-sexp(x))
                                      else parse-sexp(x)
in
    map_{i→s}(Buf_i(m,n))  ⇐  Buf_s(parse(m), parse(n))
end
```

The operation parse produces a sequence of s-expression tokens from text. This is easy to define using the ML notation for defining a function using patterns, fn *pat*.*exp* | *pat*.*exp*. If the argument is the empty list then the empty list is returned. If the argument is a nonempty list then it is parsed depending on whether it is an alphanumeric symbol or an embedded s-expression. This is a many-to-one function since information about white space and the text of the atomic symbols is lost. It makes use of two auxiliary functions, parse-an which skips over the current alphanumeric symbol and parse-sexp which skips over nested s-expressions.

The component $\texttt{Buf}_i$ in turn is linked to $\texttt{Buf}_2$, they are similar in representing the buffer as a pair of sequences of characters except that the left sequence of one is the reverse of the other.

```
map_{i→2}(Buf_i(m,n))  ⇐  Buf_2(rev(m), n)
```

The s-expression component is now integrated with the buffer. We examine here the first and most important stage—translating into $\texttt{Buf}_i$ which "bridges" the gap between s-expressions and text. Starting with the standard data transformation form,

$$\text{map}_{i\to s}(\underline{\text{move-sexp}}(\texttt{Buf}_i(m,n))) \quad \Leftarrow \quad \text{move-sexp}_s(\text{map}_{i\to s}(\texttt{Buf}_i(m,n))),$$

and assuming we have a definition for unparse, we are able to use simple syntactic manipulations (such as folding or unfolding or rewriting equals for equals using domain knowledge about sequences) to get the expression into the form:

$$\text{move-sexp}(\texttt{Buf}_i(m,n)) \quad \Leftarrow \quad \textbf{let } x \textbf{ sat } (x \mathbin{@} \text{tlsexp}(m)) = m \textbf{ in } \texttt{Buf}_i(\text{tlsexp}(m),\ \text{rev}(x) \mathbin{@} n)$$

$$\text{tlsexp}(x) \quad \Leftarrow \quad \text{unparse}(\text{tl}(\text{parse}(x)))$$

For notational convenience, we introduce "x sat eq." where the variable, x satisfies the equation (cf. unification in Prolog).

This definition is not yet complete, since we did not say how to compute unparse. Even if we did know how to compute unparse, we would not be guaranteed of getting back the original text. However, we do not want a general definition for unparse, only one where it appears in the particular context of tlsexp above.

Deriving tlsexp. A functional definition is derived for tlsexp using expression procedures. We examine the case when we apply the definition to a non-empty list. First compose parse with tl and simplify.

$$\text{tl}(\text{parse}(\text{cons}(a,x))) \quad \Leftarrow \quad \textbf{if } a \in [|\text{'a'}..\text{'z'}, \text{'0'}..\text{'9'}|] \textbf{ then } \text{parse}(\text{parse-an}(x)) \textbf{ else if } a = \text{'('} \textbf{ then } \text{parse}(\text{parse-sexp}(x)) \textbf{ else if } a = \text{')'} \textbf{ then } [\,] \textbf{ else } \text{tl}(\text{parse}(x))$$

Next compose the expression with unparse and simplify, using the simplification rule unparse(parse(x)) = x. We justify this rule by having unparse consult an "oracle," for instance, a global variable where parse has copied its argument. Later we see that we no longer need either of these operations, so we do not need to actually use the oracle.

$$\text{unparse}(\text{tl}(\text{parse}(\text{cons}(a,x)))) \quad \Leftarrow \quad \textbf{if } a \in [|\text{'a'}..\text{'z'}, \text{'0'}..\text{'9'}|] \textbf{ then } \text{parse-an}(x) \textbf{ else if } a = \text{'('} \textbf{ then } \text{parse-sexp}(x) \textbf{ else if } a = \text{')'} \textbf{ then } [\,] \textbf{ else } \text{unparse}(\text{tl}(\text{parse}(x)))$$

Abstract unparse(tl(parse(x))) into tlsexp(x).

$$\text{tlsexp}(\text{cons}(a,x)) \quad \Leftarrow \quad \textbf{if } a \in [|\text{'a'}..\text{'z'}, \text{'0'}..\text{'9'}|] \textbf{ then } \text{parse-an}(x) \textbf{ else if } a = \text{'('} \textbf{ then } \text{parse-sexp}(x) \textbf{ else if } a = \text{')'} \textbf{ then } [\,] \textbf{ else } \text{tlsexp}(x)$$

It is never actually necessary to compute unparse.

Since the s-expression component does not have complete information, it must be translated. However, we could have chosen to cache the s-expression positions as we did the newline positions, but this would be much more complicated. So we chose a quick integration process for the purposes of this example, trading off this optimization.

3.3 Summary

Putting everything together from the three examples, the process of merging the three additional components with the original system yields the new prototype in Figure 3.8.

The data structure contains the fields from the original tuple of the first three components augmented with all the fields from the page component, only the mark field from the region component (since the other fields are duplicates), and no fields from the s-expression component (since we decided on an expedient integration). The operations of the original system have been modified to operate on the new data representation. Alternative implementations of the operations of the new components have been derived for the aggregate. The results of this example are interpreted in Chapter 5 after we complete the example by adding a display in the following chapter.

```
Structure Buf_proto : BUF = struct
  type buf = Buf of
                (int×ch*          — point of editing and text in the buffer
              × ch* × ch*         — characters to the left and right of point
              × (int× int)        — the line and character position of point
              × line*             — lines in the buffer
              × int×page*         — current page and pages in the buffer
              × int)              — position of the mark
    makebuf  ⇐  Buf(0, [], [], [], ⟨0,0⟩, [], 0, [], 0)
    ...
    move-right(Buf(p,t,l,r,⟨lp,cp⟩,ts,pi,tp,m))  ⇐
      let lp',cp'  =  if (cp = #ts[lp]) then lp+1,0 else lp,cp+1,
          pi'  =  if t[p] = 'L' then pi+1 else pi in
          Buf(p+1, t, l @ [hd(r)], tl(r), ⟨lp',cp'⟩, ts, pi', tp, m)
    show-char(Buf(p,t,l,r,⟨lp,cp⟩,ts,pi,tp,m))  ⇐
      { t[p−1] | last(l) | (if (cp = 0) then 'nl' else ts[lp][cp−1]) }
    next-line(Buf(p,t,l,r,⟨lp,cp⟩,ts,pi,tp,m))  ⇐
      let d  =  (nlpos(ts))[lp] − (nlpos(ts))[lp−1],
          d'  =  npages(t[p..(p+d)]) in
          Buf(p+d, t, l @ r[..d], r[(d+1)..], ⟨lp+1,cp⟩, ts, pi+d', tp, m)
    forward-page(Buf(p,t,l,r,⟨lp,cp⟩,ts,pi,tp,m))  ⇐
      let d  =  (pagepos(tp))[pi] − (pagepos(tp))[pi−1],
          d'  =  numnl(t[p..(p+d)]) in
          Buf(p+d, t, l @ r[..d], r[(d+1)..], ⟨lp+d',cp⟩, ts, pi+1, tp, m)
    set-mark(Buf(p,t,l,r,⟨lp,cp⟩,ts,pi,tp,m))  ⇐
      Buf(p, t, l, r, ⟨lp,cp⟩, ts, pi, tp, p)
    move-sexp(Buf(p,t,l,r,⟨lp,cp⟩,ts,pi,tp,m))  ⇐
      let x sat x @ tlsexp(rev(l))  =  rev(l),
          l'  =  tlsexp(rev(l)), r'  =  rev(x) @ r,
          p'  =  #l',
          lp'  =  lp + numnl(t[p..p']),
          pi'  =  pi + npages(t[p..p']) in
          Buf(p', t, l', r', ⟨lp',cp⟩, ts, pi', tp, m)
  end
```

Figure 3.8: Adapted Buffer Prototype

Chapter 4

Reuse and Customization: Deriving an Interactive Display-Editor

In the previous chapter we constructed a buffer from components and then added additional components to adapt the buffer. We saw that components can implement parts of a datatype and that the transformation methods enabled us to integrate the parts into an aggregate data structure and to perform further optimizations. Now we change our focus to the module level and use the buffer as a part of a larger display-editor system. We see how transformation techniques are applied to hierarchically structured module systems, where modules are defined in terms of other modules. The buffer datatype that has been previously developed is reused and customized in the context of a larger interactive display-editor. Transformations are used for adapting data representations and abstract interfaces, and for optimizations.

The display editor is built in a series of stages. First we go through the exercise of adding a screen for displaying the buffer to the user. Then we generalize the system to allow multiple buffers. Finally, we introduce windows to display more than one buffer on the screen at a time. This module hierarchy is summarized at the end of the chapter in Figure 4.2.

These modules were chosen for their properties. The screen module is introduced independent of the buffer and displays some "displayable object." We then define a display view for the buffer as the screen's displayable object, and integrate it with the original buffer prototype. Once integration is achieved, additional optimization techniques are applied to take advantage of the close correspondence between the buffer and the screen. The multiple-buffers module demonstrates how module interfaces are adapted by creating a new module that includes the old one, and then propagating the operations using data transformation techniques. The multiple-windows module also demonstrates how module interfaces are adapted. In addition to the method used in the multiple-buffers module, a more synthetic approach is taken where the desired interface is exposed from existing information.

4.1 Single-Buffer Single-Window Display

This section begins by extending the text-buffer example to provide output of the buffer on a simple screen. First the data structures are introduced and then transformations are presented to integrate them. The first data structure introduced is the *screen* that displays a portion of a "display plane," some planer representation of a displayable object. The *display plane* is then designed, and defines how such an object is represented on a screen. The definition of "object" is left open at this time, as long as an object meets the requirements of the display plane, it can be shown on the screen. Finally a *display-editor* structure is designed as the tuple of the buffer and the screen and an *origin* which pins the screen to the buffer; this definition is influenced by [88]. The buffer must then be defined as the displayable object in the screen by creating a new buffer component that meets the requirements of the display plane. The first transformation then integrates this new component with the original buffer. This is done using techniques developed in the previous chapter. Then subsequent transformations optimize the display editor by more closely "coupling" the buffer and the screen.

4.1.1 Defining the Display Editor

A *screen* is a bounded portion of some displayable object and a cursor position that points at some portion of the object. Let us call the displayable representation of the object a "display plane." We then define the screen as a bounded portion of the unbounded display plane and a cursor position identifying the point of editing. This enables us to use the screen to display a variety of objects.

```
Signature SCREEN = sig
    type screen
    type origin
        disp-to-screen : origin × disp → screen
        policy : origin × disp → origin
    end
```

The operation disp-to-screen creates a screen from a portion of the display plane. The portion of the display plane to show in the screen is marked by the origin, which effectively pins the screen to the buffer. The operation policy picks an origin for the display plane.

Before defining the implementation of the screen, we define the display plane. A *display plane* provides a two dimensional representation of an object that is a useful concept for mapping different kinds of objects into a screen. We use a simple definition where the contents of the two dimensional representation are characters.

```
Signature DISP = sig
    type disp
    content : disp → planepos → ch
    current : disp → planepos
    end
```

The content operation takes a display plane and returns a function that given a display plane position, returns the character at that position, and the current operation returns the display plane position of the point of editing. We defer the implementation of the display plane at this time, until a later time when we have an object to display.

The `Screen` is represented by a cursor position and the "appearance" of the object that is displayed. The *appearance* is a function that maps a position in the screen into the corresponding character in the display. Associated with the screen is some internal state, height and width that denote the height in lines and width in characters of the physical screen. The area bounded by height and width is screensurface.

```
Structure Screen : SCREEN = struct
    type screen = Screen of cursor × (cursor → ch)
    type cursor = n × n
    type origin = n × n
    val height = 20
    val width = 80
    val screensurface = 1..height × 1..width
        project(r,c)(i,j)   ⇐   ⟨r+i, c+j⟩
        origins(d)   ⇐   {(r,c) | current(d) ∈ project(r,c)[|screensurface|]}
        disp-to-screen(o,d)   ⇐   let p = content(d) ∘ project(o)
                                      and c sat current(d) = project(o)(c)
                                      and spaces = (λi,j : N . 'sp') in
                                  Screen(c, (spaces ⊕ p)\screensurface)
        policy(o,d)   ⇐   if o ∈ origins(d) then o else (x | x ∈ origins(d))
    constraint Screen(c,a)   ⟹   c ∈ screensurface
    constraint Screen(c,a)   ⟹   domain(a)   =   screensurface
    end
```

The screen operations are defined in terms of the local operations project and origins.

```
project :   origin → cursor → planepos
origins :   disp → P(origin)
```

The project operation projects the cursor position relative to the origin to yield a display plane position. The operation origins computes a set of possible boxes of the size of the screen (represented by the top-left coordinates) that contain the point of editing in the display plane (*P*(`origin`) is the set of all origins).

The disp-to-screen operation computes a screen based on the origin and the display plane. Notice the equational nature of the computation for the cursor *c*, where it is computed such that it satisfies (sat) an equation. The appearance is constructed using content to convert the display plane into a function that takes a display plane position and returns the character at that position. In the appearance function, the display plane is first projected with the origin to yield the display-plane position. This establishes the top and left boundaries of the screen. Then the resulting display plane is restricted to the screensurface to establish the bottom and right boundaries of the screen. (The domain

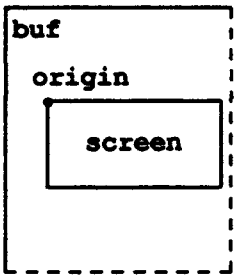


Figure 4.1: The Display Editor

restriction operator, \ maps a function f, and a subset of elements, S, to a function which agrees with f on the set S and is elsewhere undefined.) Spaces are filled into places in the screen that do not correspond to displayable text. (The functional overriding operator, $\oplus$ maps a pair of functions to one that agrees with the first, everywhere except on the domain of the second.) The policy operation returns an appropriate origin. It uses the old value if possible to minimize screen update, otherwise a new one is picked (*eg.*, such that point is in the middle of the screen).

A display editor is now constructed in terms of the buffer (developed in the previous chapter), the screen and the origin.

```
Signature DED = sig
    type ded
        ded-make :  ded
        ded-op :  (buf → buf) → ded → ded
    end
```

The display-editor data representation is defined as a tuple consisting of the buffer, origin, and screen (see Figure 4.1). The origin pins the screen to the buffer and is the position of the top-left corner of the screen relative to the buffer.

```
Structure DEd : DED = struct
    structure Buf = Buf, Screen = Screen
    type ded = DEd of (buf × origin × screen)
        ded-make() ⇐ let b' = makebuf and o' = (0, 0) in
                       let s' = disp-to-screen(o', b') in
                         DEd(b', o', s')
        ded-op(c)(DEd(b,o,s)) ⇐ let b' = c(b) in
                                  let o' = policy(o, b') in
                                    let s' = disp-to-screen(o', b') in
                                      DEd(b', o', s')
    constraint DEd(b,o,s) ⟹ s = disp-to-screen(o, b)
    end
```

The ded-op command applies the buffer operations (delete, insert, move-right, move-left, and next-line) to the buffer and updates the screen and origin appropriately. The invariant between the buffer and the origin and screen is recorded. It can be easily checked that each operation defined on the display editor preserves this invariant.

This definition of the screen is not quite correct, however. We are not able to apply disp-to-screen or policy to an object of type `buf`, it requires an object of type `disp`. In order for this definition to be correct, we must make the connection between the text buffer and the display plane.

4.1.2 Defining Buffer as a Displayable Object

Using the methodology developed in the previous chapter, we define a new buffer component that is a displayable object, that is, it implements the two operations necessary for it to be displayed on the screen. The representation of this view of the buffer consists of a sequence of lines above the point of editing, the sequence of characters to the left of point (on the line point is on), the sequence of characters to the right of point, and the sequence of lines below point. The *lines* operation returns the contents of the display plane as a single sequence of lines (of arbitrary length). The point of editing in the display plane is expressed as a line and character position. It can be thought of as a two-dimensional view of the buffer.

```
Component Buf_disp : DISP = struct
    type disp = Buf of (line* × ch* × ch* × line*)
    type planepos = n × n
        local
            lines(Buf(a,ld,rd,b))  ⇐  a @ [ld @ rd] @ b
        in
            content(d)(r,c)  ⇐  lines(d)[r][c]
            current(Buf(a,ld,rd,b))  ⇐  (1 + #a, #ld)
        end
    end
```

The representation for the new component, $\texttt{Buf}_{disp}$, must be reconciled with the previous buffer representation; this new display view must be merged with the original buffer before we are able to use it. A compatibility map between this display view and one of the previous representations is needed to relate this new type with the previous prototype buffer. Here it is natural to choose $\texttt{Buf}_2$ as the component to relate the display with, because the compatibility map is easily written.

$$\text{map}_{disp\to 2}(\texttt{Buf}_{disp}(a,ld,rd,b)) \Leftarrow \texttt{Buf}_2(\text{lines-to-chars}(a \mathbin{@} [ld]),\ \text{lines-to-chars}([rd] \mathbin{@} b))$$

There are a number of ways to accomplish the integration (see Section 3.2.1). One way is to merge the new display component with the components of the original system. This involves rederiving all the buffer operations to include the new $\texttt{Buf}_{disp}$ representation. This rederivation is not as difficult as the original derivation since the type is being adapted and

much of the derivation structure can be reused. Another way is to keep the previous buffer representation, $\texttt{Buf}_{impl}$, and translate the display operations into this representation. Since $\texttt{Buf}_{disp}$ is similar to $\texttt{Buf}_2$ this way is chosen for the sake of expediency.

The operations to be transformed are lines, content, and current. Looking at the context of where the operations are used, it is observed that lines only occurs within the context of the definition of content. Rather than computing the entire set of lines within the buffer and then taking only one character, we combine these operations to obtain an optimized version that accesses the character directly; we do this by transforming content, instead of lines. We get the following implementations:

$$\begin{aligned} \text{content}(\texttt{Buf}_i(p,t,i,nl))(r,c) &\Leftarrow t[nl[r]+c] \\ \text{current}(\texttt{Buf}_i(p,t,i,nl)) &\Leftarrow \langle i,\ p - nl[i-1]\rangle \end{aligned}$$

Since the lines operation is no longer referenced, it is *released* from the module. With these new implementations a single representation for buffer is again derived and this prototype is used with the simple display.

4.1.3 Caching the Screen

Now that we have integrated the display editor, we turn our attention to optimizations. Rather than update the entire screen after each buffer operation as the prototype just defined does, we seek to incrementally update the screen. This is accomplished by specializing the buffer and screen operations in the context of the display editor. Here they are specialized for optimizing the screen performance. One virtual model of a screen is a matrix cache; this cache is updated from the display-editor data structure. Special built-in low-level operations update the physical screen of the terminal from the cache. Operations for updating a character or line in the cache, and for efficient scrolling are provided. (These and other hardware capabilities are recorded for machines using the UNIX operating system in the terminal capability data base, "termcap").

In the display editor, there is, in effect, a narrow communication band between the buffer and the screen. We want to increase the amount of communication between the buffer and the screen. Rather than performing a buffer operation (which may be highly localized) and then mapping the entire buffer to the screen (a global operation), we wish to localize the changes to the screen whenever possible (utilizing the capabilities of the terminal to improve performance).

Incremental Update of the Screen. We express the optimization of incrementally updating the screen in terms of the data transformation scheme. We can think of the buffer and screen as different "views," where the disp-to-screen operation is the compatibility map that maps buffers into screens. Then our task is to transform a buffer operation, move-right, for example, from operating on buffers to operating on screens.

A new implementation for the move-right operation is derived by first specializing ded-op for this command.

$$\text{move-right}'(d) \Leftarrow \text{ded-op}(\text{move-right})(d)$$

Unfolding ded-op yields:

$$\begin{aligned}\text{move-right}'(\text{DEd}(b,o,s)) \Leftarrow\ & \textbf{let } b' = \text{move-right}(b) \textbf{ in}\\ & \quad \textbf{let } o' = \text{policy}(o, b') \textbf{ in}\\ & \quad\quad \textbf{let } s' = \text{disp-to-screen}(o', b') \textbf{ in}\\ & \quad\quad\quad \text{DEd}(b', o', s')\end{aligned}$$

The focus of this example is on specializing the screen. Here the screen appearance is computed by disp-to-screen each time the move-right command is invoked. The argument s, the old value of the screen is never used. But we know that if the cursor does not move off the screen, then the new appearance of the screen is exactly the same as the old value so it need not be recomputed. In order to express the updated screen in terms of its old value, we look at the definition for updating the screen. The old value is defined in terms of content(b) while the new value is defined in terms of content(move-right(b)). So, if we are able to reason about content(move-right(b)) and express it in terms of content(b), then we can reuse the old value of the screen.

The steps to accomplish this follow. The portion of move-right′ where s' is defined is shown where disp-to-screen is unfolded.

$$\begin{aligned}s' = & \textbf{let } p = \text{content}(b') \circ \text{project}(o')\\ & \quad \textbf{and } c \textbf{ sat } \text{current}(b') = \text{project}(o')(c)\\ & \quad \textbf{and } \mathit{spaces} = (\lambda i,j : N \,.\, \text{'sp'}) \textbf{ in}\\ & \quad \text{Screen}(c, (\mathit{spaces} \oplus p)\backslash\text{screensurface})\end{aligned}$$

The cursor c will always change while the appearance may stay the same in some cases so again the attention is focused on the appearance, a partial definition of which follows.

$$\text{content}(b') \circ \text{project}(o')$$

The definitions of b' and o' are expanded.

$$\text{content}(\text{move-right}(b)) \circ \text{project}(\text{policy}(o, b'))$$

Reasoning about the definition of move-right within the context of content reveals that the contents of the buffer does not change.

$$\text{content}(b) \circ \text{project}(\text{policy}(o, b'))$$

The definition of policy is next unfolded to reveal a conditional expression which is brought to the outside of the expression.

$$\textbf{if } o \in \text{origins}(b') \textbf{ then } \text{content}(b) \circ \text{project}(o) \textbf{ else } \text{content}(b) \circ \text{project}(o')$$

This reveals the expression "content(b) ∘ project(o)" which is exactly the value of the old screen appearance. The value of the screen is cached and used in this case.

if $o \in$ origins(b') then s else content(b) $\circ$ project(o')

By reasoning about content(move-right(b)), we find that under certain conditions, no change is necessary to the appearance of the screen so that we save the expense of updating it. A similar argument holds for move-left and next-line.

New implementations for the insert and delete operations can also be derived (but are not shown here) by specializing ded-op for this command. Unlike move-right the content of the screen will change, but the change is highly localized, with much of the screen remaining the same. For example, when a character that is not a newline is inserted, everything above and below the current line remains the same. Only the current line need be updated. When a newline is inserted, everything above the current line stays the same, while the lines below the current line are scrolled down by one. These observations are made in reasoning about content(insert(c,b)). By judicious manipulation, an implementation for the operation can be derived that utilizes the terminal capabilities such as inserting a character into a line and scrolling.

Screen Performance. The decisions regarding the terminal optimizations are based on the degrees of change to the screen. The types of changes that occur are, of course, dependent on the particular data structure. It is beneficial to look at how the data representation is organized. For the screen, the data structure is defined in terms of lines and characters. To assess what has changed, we need to be able to decompose the data structure in various ways so we can determine whether some piece in the updated structure is the same as some piece in the original structure. To accomplish this, we observe each data structure using its accessor functions. For the screen example, this includes functions to return a single line or some subset of lines.

How is change introduced into the screen? We start with the simplified definition for the new screen, content(op(b)). It is a function of the old display-editor buffer and observed under some context, content. What changes can we observe? Using formal manipulation, we would "ideally" like to decompose the operation, op, into smaller pieces that are either accessor functions or terminal capability functions. The accessor functions do not change the portions of the screen to which they are applied. The terminal capability functions do change the portions of the screen to which they are applied, but do so efficiently by using the specialized capabilities of the terminal. When it is not possible to decompose the operation in such a manner, the operation must perform parts of the computation to update the screen, or update the entire screen.

When must we update the entire screen and when can we make more local changes? When manipulating the operations for the display editor above, we focused on two local changes and one global change. The local changes consist of "overriding" and "offsetting" for which there exists terminal capability functions such as character insertion and scrolling. Overriding changes the screen a character or a line at a time. Offsetting shifts the characters or lines over by some amount. When inserting a non newline character, the characters to the right of the point of editing on the current line are shifted to the right and the character is inserted into the space that is opened up. The global change that we noted was updating the origin. In this case, updating the entire screen is warranted because there may be very little

in the screen that remains the same. We made no distinction for when the origin changes a small amount (less than the screen height) and a great amount. We could later specialize the former case to scroll when applicable.

4.2 Multiple-Buffers Single-Window Display

This section extends the example by adding the ability to use more than one buffer in order to learn how to adapt module interfaces by creating a new module that includes an old one, and then propagating the operations using data-transformation techniques. First a new datatype for multiple buffers is introduced and the definition of display editor is extended to make use of this new functionality. Then transformations for integrating the multiple-buffers datatype into the display editor, and for propagating the old editor definition into the new one are presented.

4.2.1 Defining a Multiple-Buffer Editor

Basic operations for a multiple-buffer editor include adding a new buffer, selecting one of the buffers for display on the screen, and deleting a buffer from the buffer list.

```
Signature MBUF = sig
    type mbuf
        make-mbsw :    mbuf
        make-buffer :  str × mbuf → mbuf
        select-buffer : str × mbuf → mbuf
        kill-buffer :  mbuf → mbuf
    end
```

Let us look to the previous definition of the simple display-editor for guidance in developing a new representation.

```
type ded = DEd of (buf × origin × screen)
```

Since we are dealing with more than one buffer, we would like to: bundle together the buffer and the origin; add a name field to identify the buffer; generalize the single entry to a list of entries; and keep the selected buffer "cached" separately from the list.

The representation we come up with is a buffer list where each entry consists of the buffer name, the actual buffer, and the origin (that pins the screen to the buffer). The current buffer (displayed on the screen) is separate from the list.

```
type mbuf = Mbuf of (str × buf × origin)* × str × buf × origin
```

This representation was chosen to simplify the presentation. Other representations are possible. For example, only the name of the current buffer could be kept separate, and then the actual buffer and origin would be looked up in the buffer list.

The operations are easily implemented using "generic" association-list operations. We use the buffer name as the key to insert, select, or remove items. We now construct a multiple-buffer display-editor.

```
Signature DED-MBSW = sig
    type ded-mbsw
        make-ded-mbsw :  ded-mbsw
        ded-op :  (buf → buf) → ded-mbsw → ded-mbsw
        make-buffer :  str × ded-mbsw → ded-mbsw
        select-buffer :  str × ded-mbsw → ded-mbsw
        kill-buffer :  ded-mbsw → ded-mbsw
    end
```

The representation for the multiple-buffer display-editor combines the buffer list with the screen.

```
type ded-mbsw = Ded-Mbsw of mbuf × screen
```

The implementations of the operations for the multiple-buffer display-editor are derived from Mbuf and DEd. The relationship between these modules is expressed in terms of a translation function and the familiar integration techniques are applied to obtain new implementations of operations based on the imported modules.

4.2.2 Integrating the Buffer-List Operations

The buffer-list operations defined in Mbuf induce corresponding operations in the multiple-buffer display-editor that are defined as data transform procedures. The relationship between the Mbuf module and the Ded-Mbsw module that includes it is expressed as:

span : mbuf → ded-mbsw
span(Mbuf(b^*, n, b, o)) ⇐ Ded-Mbsw(Mbuf(b^*, n, b, o), s)
where s = disp-to-screen(o', b)

The operations defined in Mbuf are then reimplemented in Ded-Mbsw. For example, the new definition for select-buffer is:

select-buffer'(n, span(Mbuf(b^*, n, b, o))) ⇐ span(select-buffer(n, Mbuf(b^*, n, b, o)))

After a number of transformation steps, we obtain:

select-buffer'(n, Ded-Mbsw(bl, s)) ⇐ Ded-Mbsw(bl', s')
where bl' as Mbuf(b^*, n, b, o) = select-buffer(n, bl),
s' = disp-to-screen(o, b)

In this simple case of module inclusion, the operations in Ded-Mbsw call the operations in Mbuf and then update the screen accordingly. We could continue to specialize the operations so that the screen is incrementally updated as was done in the previous example.

4.2.3 Integrating the Buffer Operations

The buffer operations from DEd (*eg.*, delete, insert, move-right) induce corresponding operations in the multiple-buffer display-editor that can be defined as data transform procedures. The relationship between the DEd module and the Ded-Mbsw module that includes it is expressed as:

```
unspan : ded-mbsw → ded
unspan(Ded-Mbsw(Mbuf(b*,n,b,o),s))  ⇐  DEd(b, o, s)
```

The fundamental operations on the buffer must be reimplemented in `Ded-Mbsw`. They are expressed in terms of the translation function and the definitions defined for the simple display-editor, `DEd`. Intuitively, the new implementations simply operates on the "cached" portion of the buffer list. The new definition for ded-op is:

```
unspan(ded-op(c,Ded-Mbsw(Mbuf(b*,n,b,o),s)))  ⇐
   DEd.ded-op(c, unspan(Ded-Mbsw(Mbuf(b*, n, b, o), s)))
```

First we unfold unspan,

```
unspan(ded-op(c,Ded-Mbsw(Mbuf(b*,n,b,o),s)))  ⇐
   DEd.ded-op(c, DEd(b, o, s))
```

and then unfold ded-op.

```
unspan(ded-op(c,Ded-Mbsw(Mbuf(b*,n,b,o),s)))  ⇐
   let b' = c(b) in
      let o' = policy(o, b') in
         let s' = disp-to-screen(o', b') in
            DEd(b', s', o')
```

Recognizing the body of unspan, we fold it,

```
unspan(ded-op(c,Ded-Mbsw(Mbuf(b*,n,b,o),s)))  ⇐
   let b' = c(b) in
      let o' = policy(o, b') in
         let s' = disp-to-screen(o', b') in
            unspan(Ded-Mbsw(Mbuf(b*, n, b', o'), s'))
```

and then take a solution.

```
ded-op(c,Ded-Mbsw(Mbuf(b*,n,b,o),s))  ⇐
   let b' = c(b) in
      let o' = policy(o, b') in
         let s' = disp-to-screen(o', b') in
            Ded-Mbsw(Mbuf(b*, n, b', o'), s')
```

The new implementation for ded-op simply operates on the selected buffer in the buffer list. Rather than choosing the general definition for ded-op, we could have chosen instead the specialized versions of the operations that optimize the screen updates.

We can think of this development process as a means to adapt interfaces. We wanted to change the interface of the simple display-editor `DEd` to include operations for dealing with multiple buffers. We adapt the interface by defining a new component, `Ded-Mbsw` and express the relationship between it and `DEd` with a translation function. Then we reimplement the operations in `DEd` into `Ded-Mbsw` using the data transformation technique. The technique provides a systematic way to propagate change.

4.3 Multiple-Buffers Multiple-Windows Display

This section extends the example by adding the ability to display more than one buffer on the screen at one time to demonstrate how to adapt module interfaces by exposing underlying representations. This involves introducing the concept of a window on the screen. Rather than introducing a new datatype, as was done in previous sections, the *expose* transformation is used to reveal the window object from the previous definition for the screen. Once the buffer and screen are decoupled, operations for multiple windows are introduced. Then the display editor is extended to include this new functionality and the previous versions propagated as was done in the previous section.

4.3.1 Defining the Multiple-Window Editor

Basic operations for a multiple-window editor include deleting a window from the screen, enlarging a window (by a line), moving the focus of attention (marked by the cursor) to another window, shrinking a window (by a line), and splitting a window into two smaller halves.

```
Signature MWIN = sig
  type mwin
    make-mbmw :  mwin
    delete-window :  mwin → mwin
    enlarge-window :  mwin → mwin
    other-window :  mwin → mwin
    shrink-window :  mwin → mwin
    split-window :  mwin → mwin
  end
```

We build on our previous work to construct a new definition. But where does the notion of window come from? Is it a new concept that must be introduced into the system, or is it somewhere hidden in the previous definition waiting to be uncovered? In the previous definition, there is an operation for mapping a display into a screen. We would like to expose more details of this operation by showing how the display could instead be mapped into a window, which is then mapped into the screen.

4.3.2 Exposing the Window

The goal of this process is to introduce a new datatype for windows. We start off with a function for mapping a display plane into a screen. We add the screen surface as an extra parameter rather than accessing it as a state variable. The idea is to reveal the underlying data representation of the data abstraction, expressing it in terms of a tuple of other representations. (The details of the *expose* transformation are explained in Section 6.2.3.)

This transformation is possible if we are able to write a function that spans these representations. The Unspan function maps a tuple representing a window and a new kind of screen into the original data representation for the screen.

```
disp-to-screen(orig, b, ss)  ⇐
    let p = content(b) ∘ project(orig)
        and c sat current(b) = project(o)(c)
        and spaces = (λi,j : N . 'sp') in
        Screen(c, (spaces ⊕ p)\ss)
```

Using the *expose* transformation, we seek to get all instances of the data abstraction to be of the form *data abs* ∘ Unspan.

```
Unspan :  (cursor × appearance) × (cursor × appearance) → (cursor × appearance)
antiproj :  origin → cursor → planepos
antiproj(r,c)(i,j)  ⇐  ⟨i − r, j − c⟩

disp-to-screen(orig, b, ss)  ⇐
    let p = content(b) ∘ project(orig)
        and c sat current(b) = project(o)(c)
        and spaces = (λi,j : N . 'sp')
        and orig' = ⟨0,0⟩
        and p' = spaces ⊕ p
        and p'' = p' ∘ antiproj(orig')
        and c' = project(c)(orig') in
        Screen(Unspan((c, p'\ss), (c', p''\ss)))
```

Replace *data abs* ∘ Unspan with unspan ∘ ⟨Window, Screen'⟩. This moves the boundary of the type inward, revealing two new abstractions in the process.

```
disp-to-screen(orig, b, ss)  ⇐
    let p = content(b) ∘ project(orig)
        and c sat current(b) = project(o)(c)
        and spaces = (λi,j : N . 'sp')
        and orig' = ⟨0,0⟩
        and p' = spaces ⊕ p
        and p'' = p' ∘ antiproj(orig')
        and c' = project(c)(orig') in
        unspan(Window(c, p'\ss), Screen'(c', p''\ss))
```

Excise unspan and then split disp-to-screen into two separate functions.

```
disp-to-screen(orig, b, ss)  ⇐  window-to-screen(disp-to-window(b, orig, ss), ⟨0,0⟩, ss)
disp-to-window(b, orig, ws)  ⇐
    let p = content(b) ∘ project(orig)
        and c sat current(b) = project(o)(c)
        and spaces = (λi,j : N . 'sp') in
        Window(c, (spaces ⊕ p\ws))
window-to-screen(orig, w, ss)  ⇐
    let p = w.appearance ∘ antiproj(orig)
        and c = project(w.cursor)(orig) in
        Screen'(c, p\ss)
```

4.3.3 Building the Display Editor

Now that the notion of windows has been revealed, we define the multiple-buffer multiple-window display-editor. The representation consists of the buffer information from the previous section on multiple-buffers single-window displays, plus a window list where each entry consists of the window name, the actual window, the name of the buffer associated with the window and an origin that pins the window to the screen. The selected window is separate from the window list.

```
type mwin = Mwin of (str × window × str × (n × n))* × (str × window × str × (n × n))
```

This representation was chosen to simplify the presentation. As with multiple buffers, other representations are possible. For example, only the name of the current window could be kept separate, and then the actual window and origin would have to be looked up in the window list.

We now construct a multiple-buffer multiple-window display-editor.

```
Signature DED-MBMW = sig
  type ded-mbmw
    make-ded-mbmw :  ded-mbmw
    ded-op :  (buf → buf) → ded-mbmw → ded-mbmw
    make-buffer :  str × ded-mbmw → ded-mbmw
    select-buffer :  str × ded-mbmw → ded-mbmw
    kill-buffer :  ded-mbmw → ded-mbmw
    delete-window :  ded-mbmw → ded-mbmw
    enlarge-window :  ded-mbmw → ded-mbmw
    other-window :  ded-mbmw → ded-mbmw
    shrink-window :  ded-mbmw → ded-mbmw
    split-window :  ded-mbmw → ded-mbmw
  end
```

The representation for the display editor combines the buffer list, the window list, and a new notion of screen.

```
type ded-mbmw = Ded-Mbmw of mbuf × mwin × screen'
```

We can propagate the operations as was done in the previous section. The implementations of the operations for the display editor are derived from Mwin and the preceding multiple-buffer display-editor. The relationship between these modules and the display editor can be expressed in terms of a translation function and the familiar integration techniques can be applied to obtain new implementations of operations based on the imported modules.

4.3.4 Integrating the Window-List Operations

The window-list operations defined in `Mwin` induce corresponding operations in the display editor that are defined as data transform procedures. The relationship between the `Mwin` module and the `Ded-Mbmw` module that includes it is expressed as:

```
span :  mwin → ded-mbmw
span(Mwin(w*, wn, w, bn, wo))  ⇐
    let s = window-to-screen(wo, w) ∘ mapwin(window-to-screen, w*) in
        Ded-Mbmw(Mbuf(b*, n, b, o), Mwin(w*, wn, w, bn, wo), s)
```

The mapwin operation recurses through the window list, applying window-to-screen to each entry in order to update the screen. The operations defined in `Mwin` can then be reimplemented in `Ded-Mbmw`. They could simply call the old operation and then update the screen appropriately.

4.3.5 Integrating the Buffer Operations

The buffer operations defined in the preceding multiple-buffer display-editor induce corresponding operations in the display editor that can be defined as data transform procedures. The relationship between the `Ded-Mbsw` module and the `Ded-Mbmw` module that includes it is expressed as:

```
unspan :  ded-mbmw → ded-mbsw
unspan(Ded-Mbmw(Mbuf(b*, n, b, o), Mwin(w*, wn, w, bn, wo), s))  ⇐
    let w' = { w | disp-to-window(o, b) },
        s' = window-to-screen((0,0), w') in
        Ded-Mbsw(Mbuf(b*, { n | bn }, b, o), s)
```

The operations on the buffer must be reimplemented in `Ded-Mbmw`. As before, the new implementation operates on the "cached" portion of the buffer list. In addition, it updates the appropriate window, and then the entire screen.

4.4 Summary

The module structure of the evolving display-editor is shown in Figure 4.2. Module inclusion is represented by solid lines between two modules. The module above is included in the module below. Translation functions are represented by arrows. Derivations are represented by dashed lines. They are numbered and refer to the items in the enumerated list below. Different kinds of adaptation include:

1. *Adding Components.* When a screen was added, the buffer did not have the necessary operations to interface with the display; two additional operations were needed. They were added by creating a new component for them, $\texttt{Buf}_{disp}$, and then integrating the component into the original system, $\texttt{Buf}_{impl}$, using the techniques for adaptation discussed in Section 3.2.

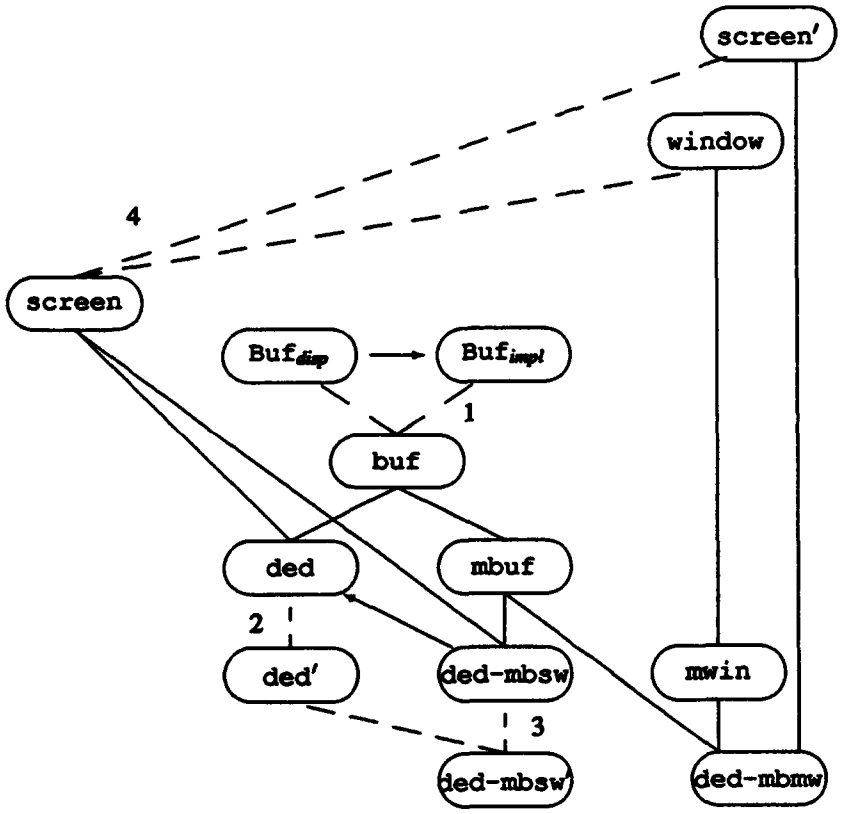


Figure 4.2: Module Hierarchy

2. *Incorporating substructure.* The buffer and screen communicate through a narrow uni-directional channel. This is accomplished by the disp-to-screen function that converts an entire buffer into a screen. It is possible to incorporate the modules for the buffer and the screen into the display-editor module, `ded`, in order to widen the channel of communication. This allows local changes in the buffer to be reflected in the screen using a form of incremental update. The channel of communication could also become bi-directional so that changes in the screen are reflected in the buffer.

3. *Extending Modules.* A series of display editors were built that progressed from having a single buffer and a single screen, `ded`, to multiple buffers and a single window, `ded-mbsw`, to one with multiple buffers and multiple windows displayed in a screen, `ded-mbmw`. As we progressed through the series of display editors, the interface for the previous editor was adapted by creating a new module that includes the old one, and then the operations were propagated using data transformation techniques.

4. *Exposing Information.* Sometimes the changes to the interface are hidden within the existing system. Instead of adding something new, a more synthetic approach is taken where the desired interface is exposed from existing information. When constructing `ded-mbmw` the concept of a `window` emerged between the abstractions for the display and the screen.

Chapter 5

Interpreting the Results of the Editor Derivation

The module interface transformation system presents an overall methodology to guide the software development process, but there are still many choices to be made by the software developer. A particular line of development was chosen in the editor example, but at certain points, other choices available to the software developer were indicated. In this chapter, the range of choices are classified at the design (Section 5.1) and implementation (Section 5.2) levels of integration. These choices are evaluated in terms of the costs they incur during the transformation process, the range of choices available at the lower levels, and the performance of the resulting implementation. The software designer will have to weigh these factors and make tradeoffs between them. Section 5.3 discusses the implications of this approach to scaling.

5.1 Integration Design Alternatives

The choices available to the software designer for integrating the collection of components into an aggregate are dependent on the properties of the components and the relationships among them. Recall that a component consists of a data structure and a collection of operations; they are related by consistency relations implemented as compatibility maps or translation functions. Components are used for constructing datatypes or objects. Now consider the integration process. The inputs to the process are the collection of components and translation functions with certain properties, and the choices made by the software developer. In Figure 5.1 we chart the integration choices made by the software developer in terms of the effects they have on the outcome, (*i.e.*, the data aggregate). Two useful measures are the level of abstraction of the data representation for the data aggregate and the time of evaluation of the translation functions and component operations in the data aggregate

Choices for the data representation appear on the "abstraction" axis. Starting at the top and moving downwards, first a single component representation is chosen, then the union of all representations, and finally the product of the representations. Below that

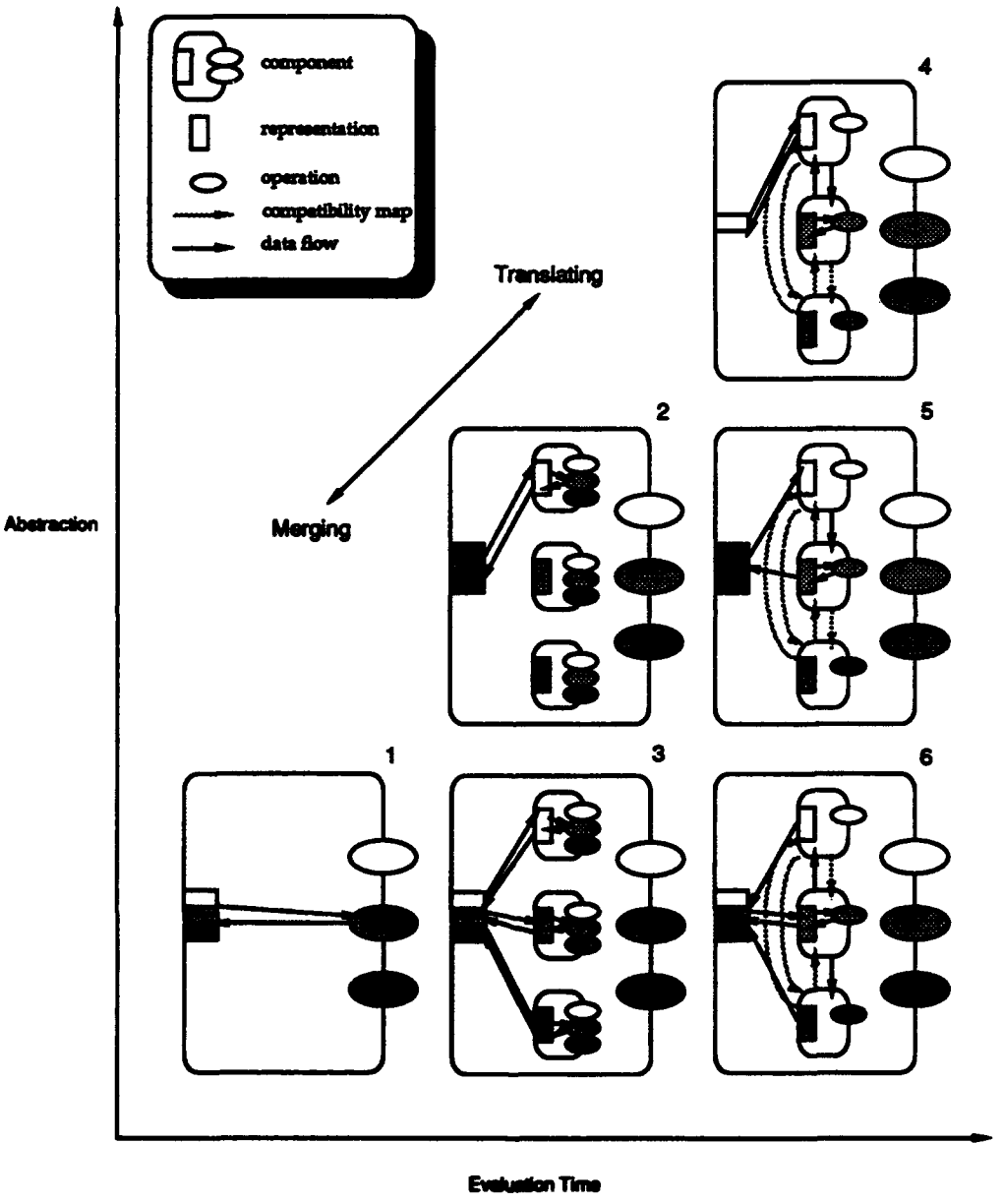


Figure 5.1: Module Interface Integration Designs.

(but not shown) could be a more generalized representation. Recall that each component operation induces an aggregate operation. One choice, then, is to use one of the component representations as the aggregate representation. The aggregate operations induced from the component operations must be defined to operate on this chosen representation. A second choice is to use all component representations in the aggregate representation. The aggregate representation could be the disjoint union or the product of the component representations. The induced operations on the aggregate must ensure the consistency of all of the component representations. If the representations of some of the components are identical then the data can be shared. Operations access the shared data representation.

Components and aggregates appear on the "evaluation time" axis. Starting at the right and moving to the left, the components at first become specialized by incorporating the translation functions, and then themselves become incorporated into the aggregate. Since each component operation induces an aggregate operation; for any operation defined on a component, a new aggregate operation can be defined that is derived or translated. A new operation may be derived that operates directly on the new representation. This is accomplished by defining the new operation in terms of the old as a data transform procedure and applying transformations to obtain an executable implementation. As an alternative, the new operation may be translated by defining it in terms of the old operation and translation functions. For example, the new operation could use translation functions to first translate the data, apply the component operation, and then translate back.

The choices made in integrating the components affect how the translation functions are used in the aggregate: (1) The translation functions are maintained and the aggregate operations dynamically evaluated. In this case the aggregate operations are translated. (2) Translation functions are partially compiled into the new implementations of the operations, but there is still some dynamic update. (3) The translation functions are compiled into the new implementations of the operations. In this case the aggregate operations are derived.

For the sake of concreteness, imagine three components, each defining a separate operation on different data representations. The operations and data representations are shaded to distinguish them. Consider the case where the operation of the middle component induces an operation on the aggregate. The numbers in Figure 5.1 correspond to the items discussed below where we pinpoint some of the choices:

1. *Incremental merging*. The translation functions and the component operation are manipulated to derive an operation on the aggregate that exploits the interdependencies among the component representations. This was the choice for the buffer prototype seen in Figure 3.5.

$$\underline{op}_{1\times 2}(span(c_2)) \Leftarrow span(op_2(c_2))$$
$$unspan(\underline{op}(Agg(c_1, c_2, c_3))) \Leftarrow op_{1\times 2}(unspan(Agg(c_1, c_2, c_3)))$$

The aggregate operation is defined as a data transform procedure where translation functions use compatibility maps and exploit the interdependencies among the component representations. Refer to Figure 5.1 at the module labeled (1). Start at the left

of the module and follow the arrows to observe how the data representation is used. In this case the aggregate operations act directly on the aggregate data representation. The representation is first accessed by an operation and then is updated with the new result.

2. *Product of the operations on the union of the data representations.* Alternative implementations of the component operation are derived for the other components. Then, depending on the current type of the aggregate, the appropriate operation on that type is selected. This alternative was not actually implemented in the buffer example, but could have been by defining the aggregate operation, op, in terms of the current type of the component.

$$\begin{aligned} op(agg) \Leftarrow\ & \textbf{case typeof}(agg)\ \textbf{is} \\ & \quad c_1.T\ \textbf{then}\ op_1(agg) \\ & \quad c_2.T\ \textbf{then}\ op_2(agg) \\ & \quad c_3.T\ \textbf{then}\ op_3(agg) \end{aligned}$$

Refer to Figure 5.1 at the module labeled (2). Again start at the left to follow how the data representation is used. The data is accessed by the appropriate component that matches its current type. The component performs the appropriate operation to update the representation with the new result.

3. *Product of the operations on the product of the data representations.* As with the product of the operations on the union of the representations, alternative implementations of the component operation are derived for the other components. But, since the data representation is now the product, all of these alternatives must be selected to update the corresponding part of the aggregate. This choice was discussed as a possibility for the buffer prototype, seen in Figure 3.3.

$$\begin{aligned} op(Agg(c_1, c_2, c_3)) \Leftarrow\ & Agg(c_1', c_2', c_3') \\ & \quad \textbf{where}\ c_1' = op_1(c_1) \\ & \quad\ \ \textbf{and}\ c_2' = op_2(c_2) \\ & \quad\ \ \textbf{and}\ c_3' = op_3(c_3) \end{aligned}$$

Refer to Figure 5.1 at the module labeled (3). The data aggregate representation, which is the product of the component representations, is projected, each component accessing the piece that corresponds to its own data representation. This is used by each component operation and the results are combined to yield a new aggregate result. In this example, the operation of the middle component is given. The alternative implementations of the component operation are defined as data transform procedures for the top and bottom components.

4. *Translating on a component representation.* One component is chosen for the data representation of the aggregate. For the sake of concreteness, consider using the first component data representation. The aggregate operation is implemented using the compatibility maps to translate to the component where the operation is defined, performing the operation, and then translating back.

$$op(agg) \Leftarrow map_{2\to 1}(op_2(map_{1\to 2}(agg)))$$

5. *Translating on the union of the data representations.* Depending on the current data representation of the aggregate, it must be translated to the component where the operation is defined before the operation is performed. There is no need to translate back, because the aggregate is the union of all components. This alternative was not actually implemented in the buffer example, but could have been by defining the aggregate operation, op, in terms of the current type of the component.

$$\begin{aligned} op(agg) \Leftarrow\ & \textbf{case typeof}(agg)\ \textbf{is} \\ & \quad c_1.T\ \textbf{then}\ op_2(map_{1\to 2}(agg)) \\ & \quad c_2.T\ \textbf{then}\ op_2(agg) \\ & \quad c_3.T\ \textbf{then}\ op_2(map_{3\to 2}(agg)) \end{aligned}$$

6. *Translating on the product of the data representations.* This choice for implementing the aggregate is inspired directly by the aggregate definition seen in Figure 3.2. With compatibility maps implementing the consistency relations, then the aggregate operation is defined by extracting the component representation from the product, performing the operation, and then using the compatibility maps to update all portions of the product.

$$op(agg) \Leftarrow proj_2^{-1}(op_2(proj_2(agg)))$$

The projection extracts the appropriate component; the inverse of the projection is defined in terms of the compatibility maps to produce the aggregate.

The choice for $\texttt{Buf}_{proto}$ is at (1) in Figure 5.1 but alternatives (3) and (6) were also considered in the discussion. Merging the pages and regions components also are at (1). Translating the pages, s-expressions, and display components are at a position higher up at the top-left in the diagram.

The integration alternatives of merging or translating with the components of the original system or the implementation can also be seen in this diagram. Translating is near the top and to the right, where only one of the component representations is chosen. This requires the implementations for the operations on the other components be translated. Moving downwards and to the left shows the alternatives for merging. At the bottom left, since all of the component data representations appear in the aggregate, and the translation functions have been incorporated into the operation definitions, there is no other alternative than to ensure that each operation updates the aggregate. There are a continuous set of choices, the alternatives are not restricted to the discrete set shown in the figure, which highlight some of the interesting choices.

5.2 Integration Implementation Alternatives

It is desirable that the software developer understand the costs associated with the various alternatives. This section first discusses the cost measures available for making this analysis and then looks at the choices and costs made in the display-editor derivation.

5.2.1 Cost Measures

What criteria about the cost of integration is available to make the various decisions to implement the aggregate? One way to measure the cost of integrating a collection of components is to count the number of data transform procedures that must be defined (which is proportional to the number of data transformation steps applied). The number of definitions that must be added for each component with n operations is:

$$n \text{ operations} \times m \text{ merges/operation}$$

Considering the collection of components and compatibility maps that connect them as a graph, "merges" is the sum of the length of the paths between the component being merged and the other components. Duplicate paths are factored out. For example, referring to Figure 3.7, this is 2 for $\texttt{Buf}_2$ since it must reach $\texttt{Buf}_3$ through $\texttt{Buf}_1$.

The cost of adding a component to an existing system of integrated components is also measured in the the number of data transform procedures that are defined. Given a core system with m components and l operations and a new component with n operations, then merging with the original components may add up to m (data transform procedure) definitions for each new operation. The number of components, m, serves as an upper bound on the number of merges required. In addition, there are l definitions to update the existing system, giving a total of $n \times m + l$ definitions. Translating into the original system adds up to m definitions for each new operation, the original system remains unchanged.

The number of definitions is directly proportional to the number to transformation steps necessary to implement a functional prototype. Based on experience, there are on the order of 10 derivation steps necessary to transform a data transform procedure into a functional definition. Approximately 10 percent of these steps are insight steps. (See Appendices C and D for examples.) More experience clearly is needed to infer the number of steps for general problems. It is useful to make this distinction between the insight (or "eureka") steps and the others because the former require manual assistance from the software developer whereas most of the steps of the latter can be automated. It is difficult to quantify the effort required by the software developer in absolute terms. Thus the scenarios below discuss the relative costs. Although there may be more steps in some cases, the effort required by the software developer may be less than the other alternatives with fewer steps because the problem is broken into smaller conceptual pieces that are easier to reason about. Often, the amount of work by the software developer is greater initially (especially in a new problem domain), but as the derivation progresses, more information is gathered that can be reused; thus the cost is amortized over the duration of the derivation. The cost of integrating the component depends on the integration alternative.

Merging with the Original System. In merging with the original system, new implementations are derived for operations of the new component (requiring $n \times m$ definitions). New implementations must also be derived for the l operations of the existing components. The existing derivations for the old implementations of the operations can be reused directly. The derivation structure for the new operations may be similar to that of the old operations

so that insights may be reused as well. The specialization step requires transforming the sum of all the operations, $l + n$, and may reuse information about specializing the original system.

There are more steps involved in merging the new component with the original system than in the other alternatives, but they may be simpler, requiring less user insight, since more information is available. Since there is more information, more optimization choices are available. The representation of the specialized aggregate may be different from the original system. For example, in the buffer implementation, we kept $\texttt{Buf}_1$ and deleted $\texttt{Buf}_2$, but we may wish to reverse the decision if the new component is used frequently and is more efficient in $\texttt{Buf}_2$.

Translating into the Original System. In translating the new component into the original system, new implementations are derived for operations of the new component (requiring $n \times m$ definitions). Since the original system does not change, new implementations do not have to be derived for the operations of the existing components. The specialization step requires transforming the sum of the operations, but since the original system does not change, the information about specializing the original system can be reused directly. The cost, n, is incurred in specializing the operations of the new component.

Merging with the Implementation. When merging the new component with the implementation instead of the original system, then the number of components, m, is no longer a useful measure for the upper bound of merges required because many of the components may be specialized or eliminated in the aggregate implementation. Therefore, $f(m)$ is used instead to indicate the affect of the specialization step. In the buffer implementation example, f reduced m about 50 percent because the prototype aggregate consisted of $\texttt{Buf}_1$, $\texttt{Buf}_2$, and $\texttt{Buf}_3$ and the implementation step deletes $\texttt{Buf}_2$ and simplifies $\texttt{Buf}_3$ so that there is less work in merging with the aggregate.

There are fewer steps involved in merging with the implementation than in the alternatives discussed above, however, they may be more complex since the "natural" representation of the component may have been specialized in the aggregate so that it is more efficient but less easy to manipulate during integration. Still, the new component is available for optimization. When the most "natural" representation for the new operations is the new component, the specialized aggregate may not prove to be that great of a hindrance. There are fewer choices available to the software designer because of the more specialized data representation and operations than in the alternatives discussed above. But the greater number of choices in the other alternatives may lead to unnecessary steps where the results are eliminated later in the process. For example, merging the $\texttt{Buf}_2$ component only to eliminate it in the specialization step.

Some of the information in the derivation of the original system can be reused. What is new is the relationship between the new component and one existing component. Once that is "bridged," computing the relationship of the new component to the rest of the system can make use of previous derivation (*i.e.*, insight steps, merging process, and interrelationships).

Translating into the Implementation. When translating the new component into the implementation, new implementations of the operations of the new component need only be derived (requiring $n \times f(m)$ definitions). There are no changes needed to the existing system. This method has the fewest steps of all of the alternatives, but they have the potential to be the most complex since the software designer has the least flexibility (because there is less information available) and the operations must be translated into one and only one representation. There is only one choice for the data representation, keeping the existing data aggregate representation. As with the other alternatives, there is the potential for some reuse of the existing derivation structure.

5.2.2 Integrating Components

Choices. As the software designer, we made particular choices in defining the prototype edit-buffer. For the pages component, we set up definitions to either merge the pages component with the original system or to translate it into the original system. Since the regions component adds new information, it must be merged. Since there are no dependencies between the mark and the existing system it is easy to add the component at either level of original components or implementation. Since the s-expression component does not have complete information, it must be translated. However, we could have chosen to cache the s-expression positions as we did the newline positions, but this would be much more complicated. So we chose a quick integration process for the purposes of this example, trading off this optimization.

Cost. Recall that the original buffer system consisted of three components and seven operations. Derivations were done at the aggregation level (transforming the aggregate into a prototype) and on the implementation level (transforming the prototype into an efficient implementation). The cost is measured in the number of data transform procedures that must be defined in order to integrate the collection of components. At the aggregation level there are 14 definitions.

	n operations	m merges	$n \times m$
$\texttt{Buf}_1$	3	2	6
$\texttt{Buf}_2$	3	2	6
$\texttt{Buf}_3$	1	2	2
Total			14

At the implementation level there are 7 definitions. Using on the order of 10 derivations steps for each definition, the total number of steps is approximately 210, with 20 insights. The insights are used for translating between the domains of the data representations. They are often shared so there are in fact fewer novel insight steps that the developer must come up with.

Adapting the buffer by adding a component for pages introduced one new component with three operations. The original system had three components with seven operations. This yields 16 definitions that are transformed to obtain the integrated prototype.

	n operations	m merges	$n \times m$
$\texttt{Buf}_p$	3	3	9
$\texttt{Buf}_1$	3	1	3
$\texttt{Buf}_2$	3	1	3
$\texttt{Buf}_3$	1	1	1
Total			16

Translating, on the other hand, yields 9 definitions that are transformed to obtain the integrated prototype because the existing operations for $\texttt{Buf}_1$, $\texttt{Buf}_2$, and $\texttt{Buf}_3$ are not affected.

5.2.3 Integrating Modules

Choices. Similar choices between merging and translating using the original system or derived implementation exist at the module level as at the component level. As the software designer, we made particular choices in defining the prototype display-editor. For the display component, we chose to translate the component into the existing buffer implementation. When adding multiple buffers and multiple windows we chose to translate the list and buffer components into the display editor in order to propagate the operations into the module that imports them.

Cost. The cost of using this approach at the module level is similar to the cost at the component level and can be measured in the number of data transform procedures that must be defined in order to integrate the collection of modules. In the example of screen caching, we are in effect defining alternative implementations for the buffer operations in the screen component, so the cost is that of transforming a data transform procedure into a functional definition. Integrating the list operations or a simpler display-editor into a more complex display-editor that adds additional functionality are simplified forms of the data transform procedure. Since the aggregate representation includes the component representation, the new aggregate operation simply uses the old component operation to update the appropriate fields.

5.3 Scaling

The benefits to scaling occur primarily at the integration design level. Complexity is managed through abstraction, modularization, and step-wise transformation. The focus of the software designer is on the design domain. These design decisions are translated into changes throughout the system at the integration implementation level to integrate and optimize the system. The formal manipulations at this level are generally carried out within local contexts. However, in order to claim that this method truly scales, automated assistance is needed at the integration implementation level in carrying out all of the steps. Most of these are mechanical steps that could be performed with automated support.

5.3.1 Components

How does the methodology scale as the number of components increases? The examples of adding pages, regions, and s-expressions adapt the system by adding more definitions to the set that constitute the existing system, and then applies the derivation process all over again. Only a single connection is needed between the new component and one of the existing components or the data aggregate. This facilitates adaptation, since, when a new component is added, it is not required to define every possible connection to all of the existing subcomponents. It should be sufficient to define a single connection between the new component and the aggregate. Only a single compatibility map is needed (and not its inverse). This aids the software-developer when the inverse is difficult to define or the compatibility map is not one-to-one. Of course, the other interconnections are derived during the integration process, but here is where support from automation and reuse can be provided. Also, since we are dealing with datatypes, it is highly likely that there will be a limited number of components that make up the datatype.

Can the existing system be considered a component as one way to scale up? This possibility was mentioned briefly when we discussed merging or translating a new component with the implementation (Section 2.1.1 and Section 5.2.1). We can treat the aggregate as a component when there are no interdependencies among the fields of the aggregate. When there are interdependencies, then these must be taken into account to preserve the internal consistency of the aggregate.

5.3.2 Modules

How does the methodology scale as the levels of module hierarchies increases? The benefits of scaling come from using a module system, in effect "scaling down." Modules help us to scale down by limiting the focus to one module at a time and its interconnections. Perhaps, just as the number of components will be limited since they comprise a datatype, the number of modules may be limited if they comprise idioms of a higher-level of abstraction [80].

Chapter 6

A Framework for the Module Transformation System

Now that we have seen an example that demonstrates the derivation process and techniques, we return to the module transformation system introduced in Chapter 2 and examine a more rigorous description of the process. Section 6.1 describes the steps involved in using module transformations in software development. Terms used in the example such as "component," "consistency relation," "aggregate definition," "prototype," and "implementation" are given a precise meaning and the process of obtaining efficient implementations from a collection of components is formalized. Section 6.2 describes the module transformation rules and demonstrates the steps in applying them. The findings are summarized in Section 6.3 which describes what was added to the framework and why. Providing a framework enhances the understanding of the terms used informally, provides structure to aid the software designer in using the approach and is an important step towards automating the system.

6.1 Module Transformation in Software Development

This section provides a more rigorous explanation of the software development strategy described in Section 2.4 and demonstrated in Section 3.1. The first three phases of the process: program design, program composition, and component aggregation, are explained using a simple theory based on algebraic specification that provides a precise meaning to constructing an aggregate specification in terms of components. Once a precise meaning of the specification is given, it is manipulated in the subsequent aggregate integration phase of the process to produce an aggregate definition. Reflecting on how the specification is manipulated gives insight into constructing an algorithm that automates the process. The final two phases of aggregate implementation and optimization refine the definition using the module transformation rules in Section 6.2.

As stated earlier, a notation based on Standard ML modules [60] is used to represent datatype definitions. In addition to representing datatype definitions, the notation needs to also express the other structures in the transformation process. These notations are introduced as they are needed and summarized at the end of the chapter in Section 6.3.

6.1.1 Program Design

To start with, we need a method for defining the datatype of interest, for example, the text buffer. Algebraic specifications are especially appropriate because abstract datatypes are treated as algebras. Treating datatypes as algebras is useful for: (1) proving properties about the datatype such as consistency; (2) showing the correctness of an implementation with respect to a specification; and (3) using logic to rewrite (or simplify) equations by substituting an expression with an equivalent one. This does not mean that the software designer must always start with a formal specification. In actual engineering practice, the software designer might use a high-level prototype rather than writing a specification. We in fact followed this approach in the development of the interactive display-editor, since the focus of the software development method is on integrating software components which takes place at the system design level. In this chapter, we start with a specification to motivate the method and to enable us to give a precise meaning to the combination of software components into an aggregate datatype.

6.1.2 Program Composition

The first subset of operations to be considered is the *constructor set* [37], which has the property that all instances of the datatype (*i.e.*, all terms in the algebra) are generated by using only constructor set operations. We start by defining the abstract interface. An *abstract interface* is simply a signature. We use a syntax similar to the Standard ML `signature` declaration to specify abstract interfaces. For example, in the datatype `buf`, the operations createbuf, ins, and point constitute a constructor set.

```
Signature BUF_c = sig
    type buf
        createbuf : buf
        ins :  ch × buf → buf
        point :  buf → buf
    end
```

This gives an operation for creating the buffer, adding a new character into the buffer, and setting the focus of editing. A constructor set is used to define what I call the *core component*.

Using the framework and terminology of algebraic specification of abstract datatypes [32], an S-sorted signature, Σ, is defined for the text buffer operations (where sorts correspond to types). The signature consists of the names and functionalities of operations over the sorts in the sort set S. Given this signature, the semantics of the text buffer operations is defined by writing an algebraic specification $\langle S, \Sigma, E \rangle$, where Σ is the S-sorted signature and E is a set of Σ-equations.

> **Definition 1.** A *core component specification* is an algebraic specification $\langle S_{core}, \Sigma_{core}, E_{core} \rangle$. S_{core} is the sort of interest, Σ_{core} is a signature of a constructor set of operations, and E_{core} is empty.

Notice that a core component specification, since it specifies only constructors, generally does not have any equations. This is a rather "loose" specification of a buffer since it does not express how the buffer is initialized or the relationship between the focus of editing and the text. Such design decisions are deferred to a later time when the core component is used as the basis for defining other components. The core component is not meant to be implemented, its purpose being to provide a notion of "equivalence," that is, what it means for other components to be views or alternative implementations of the same datatype.

Once a core component is specified, we obtain other components by supplementing the core component specification with additional operations.

> **Definition 2.** A *component specification* is an "enrichment" of a core component specification.

As described in [37], an enrichment of a specification is obtained by adding new operations along with "axioms" that define the behavior of each new operation. This is always a "strict" extension: (1) it is non-empty; (2) since E_{core} is empty, there is no change to the properties of the existing operations; (3) none of the existing operations are taken away.

For example, we enrich the buffer core component by introducing the new operations, move-left, move-right, and show-char to obtain functionality for moving the cursor to the left or to the right, and for showing the character at the cursor position. The signatures of the operations are listed first, followed by axioms that define the meaning of the new operations in terms of the core component operations.

```
Signature BUF1 = sig
  structure B :  BUFc
    move-left :  B.buf → B.buf
    move-right :  B.buf → B.buf
    show-char :  B.buf → ch
    axiom move-left(B.insert(c, b))  =  move-left(b)
    axiom move-left(B.point(b))  =  b
    axiom move-right(b)  =  B.point(b)
    axiom show-char(B.ins(c, b))  =  show-char(b)
    axiom show-char(B.point(B.ins(c, b)))  =  c
    axiom show-char(B.point(B.point(B.insert(c, b))))  =  show-char(B.point(b))
  end
```

> **Definition 3.** A *component implementation* is an implementation of a component specification. Let P be a specification. Then the particular way that the implementation I satisfies P is described by the "view," $v : P \Rightarrow I$.

As in [33], a view consists of a mapping from the sorts of P to the sorts of I, and a mapping from the operations of P to the operations of I.

For example, we provide an implementation for the BUF_1 component specification, defining buf as a data structure consisting of an integer (representing the cursor or point of editing) and a sequence of characters (representing the text). The operations are defined on

this data structure. We use a notation similar to the Standard ML `structure` declaration except that we call it a `component` because, unlike a structure, not all of the operations of the signature need be implemented.

```
Component Buf1 : BUF = struct
  type buf = Buf of (int × ch*)
    move-left(Buf(p,t))   ⇐  Buf(p − 1, t)
    move-right(Buf(p,t))  ⇐  Buf(p + 1, t)
    show-char(Buf(p,t))   ⇐  t[p − 1]
    constraint Buf(p,t)   ⟹  0 ≤ p ≤ #t
  end
```

The view shown below defines how the implementation $\texttt{Buf}_1$ satisfies the specification $\texttt{BUF}_c$. We extend the notation to include a definition for views taken from OBJ3 [33] and adapted to ML syntax. First the sort in the core buffer is mapped into the corresponding sort in the component. Then each of the operations in the core is mapped into the corresponding operations in the component.

```
view V1 from BUFc to Buf1 is
  sort buf   to  Buf(int × ch*)
  vars c :   ch
    op createbuf   to  Buf(0, [ ])
    op ins(c, _)   to  let Buf(p,t) = _ in Buf(p, t[..(p − 1)] @ [c] @ t[p..])
    op point(_)    to  let Buf(p,t) = _ in Buf(p + 1, t)
  endv
```

The subsequence of the sequence s from the first to the i^{th} element is denoted $s[..i]$; the subsequence of the sequence s from the i^{th} to the last element is denoted $s[i..]$. The placeholder for the sort of interest (in this case `buf`) is denoted _. This next view defines how the implementation $\texttt{Buf}_2$ satisfies the specification $\texttt{BUF}_c$.

```
view V2 from Bufc to Buf2 is
  sort buf   to  Buf(ch* × ch*)
  vars c :   ch
    op createbuf   to  Buf([ ], [ ])
    op ins(c, _)   to  let Buf(l,r) = _ in Buf(l, [c] @ r)
    op point(_)    to  let Buf(l,r) = _ in Buf(l @ [hd(r)], tl(r))
  endv
```

The consistency relation provides a correspondence between the data objects manipulated in one implementation with those in another.

Definition 4. Let P and Q be component specifications that are enrichments of some common core component specification C. Let I (with view v_I) and J (with view v_J) be implementations of P and Q respectively. Then there is a component *consistency relation* between the terms in the alternative implementations: $v_I(t)$ map_I^J $v_J(t)$, where t is a Σ-term in the core component.

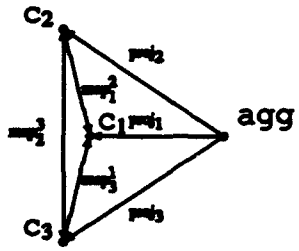
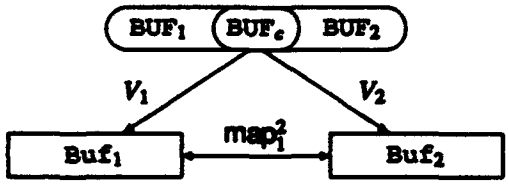


Figure 6.1: Projections of the Aggregate.

The views V_1 and V_2 specify this consistency relation between the $\texttt{Buf}_1$ and $\texttt{Buf}_2$ components (defined in Section 3.1).

In the figure below, we see how all of the pieces fit together. The intersection of the specifications for $\texttt{BUF}_1$ and $\texttt{BUF}_2$ is the core specification $\texttt{BUF}_c$. The consistency relation is defined implicitly in terms of the views V_1 and V_2.



The consistency relations provide a notion of consistency for the aggregate data objects.

6.1.3 Component Aggregation

Suppose we have a specification P, a core component c_0, and a collection of component implementations $c_1, \ldots, c_n$, where $\Sigma_{c_1} \cup \cdots \cup \Sigma_{c_n} = \Sigma_P$ (*i.e.*, every operation of P is implemented in some component). Further, for each pair c_i, c_j, with $1 \leq i, j \leq n$ and $i \neq j$, we have that $\Sigma_{c_i} \cap \Sigma_{c_j} = \Sigma_{c_0}$ (*i.e.*, each component defines distinct subsets of the operations). Then an *aggregate* is formed from the collection of components to provide a refinement of specification P.

> **Definition 5.** An *aggregate specification* refines P by specifying an aggregate data representation constructed from the data representations of the components $c_1, \ldots, c_n$. This aggregate representation has the following properties:
>
> - There are projection functions, proj_i, which map an aggregate data object to an object of component c_i. See Figure 6.1 where $n = 3$.
> - Every operation op_i in component c_j induces a corresponding operation op on the aggregate, a. The operation op satisfies the following equations:

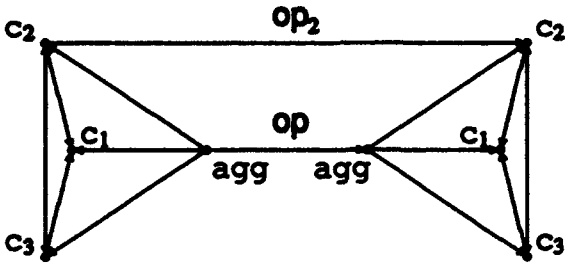


Figure 6.2: Aggregate Operation Definition.

$$\text{proj}_j(\text{op}(a)) = \text{op}_i(\text{proj}_j(a))$$
$$\forall k \in (\{1,\ldots,n\} \setminus j)\ \text{proj}_j(a)\ \text{map}_j^k\ \text{proj}_k(a) \implies \text{proj}_j(\text{op}(a))\ \text{map}_j^k\ \text{proj}_k(\text{op}(a))$$

See Figure 6.2 where $n = 3$, $i = 2$, and $j = 2$. Each operation might actually have arguments other than the data object.

Notice that an aggregate specification, while it is based on component implementations, does not quite constitute an implementation of the specification P, due to the use of consistency relations. Thus, we say that an aggregate is a *refinement* of P.

Continuing our example and taking move-right to illustrate the point, recall the axioms for the move-right operation in the text buffer example.

$$\text{axiom}\ \text{proj}_1(\text{move-right}(b)) = \text{move-right}_1(\text{proj}_1(b))$$
$$\text{axiom}\ \text{proj}_1(b)\ \text{map}_1^2\ \text{proj}_2(b) \implies \text{proj}_1(\text{move-right}(b))\ \text{map}_1^2\ \text{proj}_2(\text{move-right}(b))$$
$$\text{axiom}\ \text{proj}_2(b)\ \text{map}_2^3\ \text{proj}_3(b) \implies \text{proj}_2(\text{move-right}(b))\ \text{map}_2^3\ \text{proj}_3(\text{move-right}(b))$$
$$\text{axiom}\ \text{proj}_3(b)\ \text{map}_3^1\ \text{proj}_1(b) \implies \text{proj}_3(\text{move-right}(b))\ \text{map}_3^1\ \text{proj}_1(\text{move-right}(b))$$

The projection proj_1 maps the aggregate to $\texttt{Buf}_1$. The definition of move-right on the aggregate is defined in terms of the operation, move-right_1, which operates on the component $\texttt{Buf}_1$. The remaining axioms ensure consistency.

6.1.4 Aggregate Integration

As an intermediate step towards obtaining an implementation of the specification P, the axioms that define the aggregate specification are manipulated (*eg.*, using rewrite rules) to obtain expression procedures for spanning data representations.

Definition 6. An *aggregate definition* is a refinement of an aggregate specification. The data representation is defined as the product of the component data representations and the operations are defined in terms of the component operations as data transform procedures.

We saw these procedures in the buffer definition in Figure 3.4 and extract the definition for move-right below.

$$
\begin{array}{l}
\textbf{local} \\
\quad \text{unspan}_c(\texttt{Buf}(p,t,l,r,\langle lp,cp\rangle,ts)) \Leftarrow \\
\qquad \texttt{Buf}_1(\{\, p \mid p_2 \mid p_3 \,\},\ \{\, t \mid t_2 \mid t_3 \,\}) \\
\qquad\quad \textbf{where } \texttt{Buf}_1(p_2,t_2) = \text{map}_{2\to 1}(\texttt{Buf}_2(l,r)) \\
\qquad\qquad \textbf{and } \texttt{Buf}_1(p_3,t_3) = \text{map}_{3\to 1}(\texttt{Buf}_3(\langle lp,cp\rangle, ts)) \\
\textbf{in} \\
\quad \text{unspan}_c(\underline{\text{move-right}}(b)) \Leftarrow \text{move-right}_1(\text{unspan}_c(b)) \\
\textbf{end}
\end{array}
$$

Indeed, this process can be automated. An algorithm for generating an aggregate definition from a collection of components and "compatibility maps" is given later in this section. The definitions for the operations of the aggregate are in the form of "data transform procedures."

> **Definition 7.** A *compatibility map* is a function that respects the consistency relation. It translates one component representation into another representation.

Depending on the consistency relation, it may not be possible to implement compatibility maps in both directions, but normally it is straightforward to implement one of them. When a compatibility map exists from component c_i to c_j, we say that c_j can be *reached* from c_i, or that c_i can *reach* c_j. Here we extract the compatibility map from $\texttt{Buf}_2$ to $\texttt{Buf}_1$ (Figure 3.4).

$$\text{map}_{2\to 1}(\texttt{Buf}_2(l,r)) \Leftarrow \texttt{Buf}_1(\#l,\ l \mathbin{@} r)$$

We define alternative implementations on data representations with "data transform procedures," in terms of the original implementations and "translation functions."

> **Definition 8.** A *translation function* is a function that translates one data representation into another representation.

The spanning functions, span, and its inverse, unspan, and compatibility maps are all examples of translation functions.

> **Definition 9.** A *data transform procedure* defines an alternative implementation and may take one of two forms:
>
> 1. Given a program f using a data representation D and an injective function, span, that translates elements of the data representation D to elements of the data representation D', we define f′ as:
>
> $$f'(\text{span}(d)) \Leftarrow \text{span}(f(d))$$
>
> (with universal closure over occurrences.)

NODES = the set of all the components.
MAPS = the set of all the compatibility maps.

foreach c that is a component
  let $C = \{c\}$, $N = NODES - C$ **in**
    foreach *op* that is an operation defined in the component c
      while $N \neq \emptyset$
        $\exists j \in C, f \in N, \text{map}_{j \to f} \in MAPS \implies$
          $C' = C \cup \{f\}$; "output data transform using span"; $C = C'$; $N = N - \{f\}$
        $\exists j \in C, f \in N, \text{map}_{f \to j} \in MAPS \implies$
          $C' = C \cup \{f\}$; "output data transform using unspan"; $C = C'$; $N = N - \{f\}$

Figure 6.3: Generating Aggregate Definitions

2. If, instead, there is a surjective function, unspan, that translates elements of the data representation D' to elements of the data representation D, we define f′ as:

$$\text{unspan}(f'(d)) \Leftarrow f(\text{unspan}(d))$$

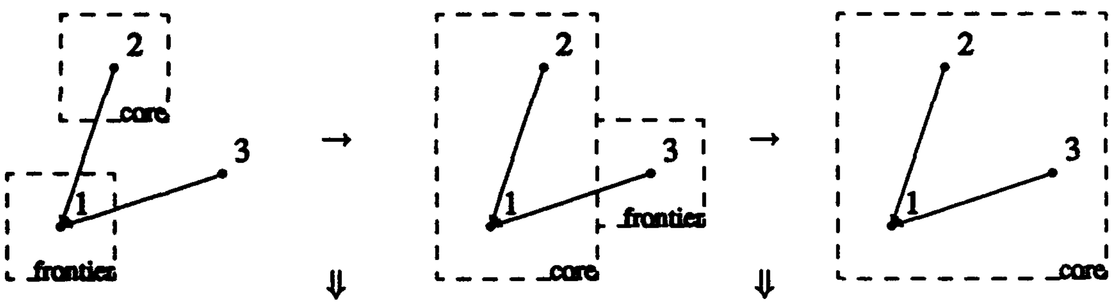
Notice the similarity with Definitions 1 and 2 presented in Section 2.2 (where the domains are now datatypes). This form of definition is called an "expression procedure" in [74, 75] since an expression appears on the lefthand side of the procedure definition. These two definitions give meaning to f′ only when it is applied to the result of span, or when unspan is applied to its result. We rely on applying syntactic transformations to obtain a functional definition for the program f′ on the data representation D'. The span function must be injective. Otherwise f′ may not be able to distinguish distinct values that f could. The unspan function must be surjective. Otherwise there could be some values defined on f but not on f′; therefore, f′ would not be a valid implementation of f because it could not handle all values that f could. Data transform procedures are used to explain the module transformation rules that affect data representations, (*i.e.*, *shift*, *translate*, and *expose*).

Now we examine the algorithm for generating an aggregate definition and see how it preserves the properties of the axioms.

An Algorithm for Generating Definitions

The algorithm for generating definitions in Figure 6.3 is based on directed graph connectivity where nodes of the graph are merged with the application of the integration transformations. Consider the collection of components and compatibility maps as a directed graph, where the components are nodes and the compatibility maps are directed arcs. Each operation of every component is considered in turn with the goal of producing the corresponding aggregate operation. The node representing the component under consideration constitutes the *core*. All nodes that are connected to some node within the core (via an arc) constitute the *frontier*. The nodes in the frontier are "coalesced" with

the core by expanding the core to include the nodes in the frontier and by producing the aggregate operation definition using a variant of the data transformation technique. This is done separately for nodes in the frontier that can be reached from a node in the core, and for nodes in the frontier that can reach a node in the core, since different data transformation definitions are required. Then a new frontier is defined based on the expanded core and the process of coalescing connected nodes is repeated until all of the nodes comprise the core. Then the whole process is repeated until all the operations of all the components are reimplemented in the data aggregate. Here we see the process for makebuf defined in component $\texttt{Buf}_2$. The aggregate operation definitions that are produced are shown below the graph as it is coalesced.



$$\begin{aligned}&\text{makebuf}_{1\times 2} \Leftarrow \text{span}(\text{makebuf}_2)\\ &\text{span}(\texttt{Buf}_2(l,r)) \Leftarrow\\ &\quad \texttt{Buf}_{1\times 2}(p,\ t,\ l,\ r)\\ &\qquad \textbf{where } \texttt{Buf}_1(p,t) =\\ &\qquad\qquad \text{map}_{2\to 1}(\texttt{Buf}_2(l,\ r))\end{aligned}$$

$$\begin{aligned}&\text{unspan}(\text{makebuf}) \Leftarrow \text{makebuf}_{1\times 2}\\ &\text{unspan}(\texttt{Buf}(p,t,l,r,\langle lp,cp\rangle,ts)) \Leftarrow\\ &\quad \texttt{Buf}_{1\times 2}(\{\ p \mid p_3\ \},\ \{\ t \mid t_3\ \},\ l,\ r)\\ &\qquad \textbf{where } \texttt{Buf}_1(p_3,t_3) =\\ &\qquad\qquad \text{map}_{3\to 1}(\texttt{Buf}_3(\langle lp,cp\rangle,\ ts))\end{aligned}$$

The data transform templates (shown below) are used to generate the "code" for the aggregate definition operations. The fixed "code" in the templates is in **bold face**. Placeholders (*eg*., *op*, to be filled in with the operation name) are in *italics*. Once instantiated, the templates produce the data transform procedure definitions. The auxiliary function Π takes a set and returns a literal tag composed from its elements. This is used as a unique and descriptive subscript for the intermediate aggregates that are built as the nodes of the graph are coalesced. The function Φ takes a set and builds a literal parameter list out of the elements. This is used to create unique variable names for the parameters of the aggregate.

$$\begin{aligned}&\textbf{local}\\ &\quad \textbf{span}(\textbf{Agg}_{\Pi(C)}(\Phi(C))) \Leftarrow \textbf{Agg}_{\Pi(C')}(\Phi(C'))\\ &\qquad\qquad\qquad\qquad \textbf{where } f = \textbf{map}_{j\to f}(j)\\ &\textbf{in}\\ &\quad op_{\Pi(C')}(\textbf{span}(\textbf{Agg}_{\Pi(C)}(\Phi(C)))) \Leftarrow \textbf{span}(op_{\Pi(C)}(\textbf{Agg}_{\Pi(C)}(\Phi(C))))\\ &\textbf{end}\end{aligned}$$

Data Transform Template - Span.

When an arc (and thus a compatibility map) exists from a node in the core to a node in the frontier, then **span** is used to map the data representation of an aggregate consisting of

the nodes of the core into an aggregate consisting of the nodes of the core and the frontier. The span function uses the compatibility maps to define how the data representations of the nodes in the frontier are generated from the data representation of a node in the core.

local
$\quad$ **unspan**($\mathbf{Agg}_{\Pi(C')}(\Phi(C'))$) $\Leftarrow$ $\mathbf{Agg}_{\Pi(C)}(\Phi(C[f \backslash \{ j \mid f \}]))$
$\qquad$ **where** f = $\mathbf{map}_{f \to j}(f)$
in
$\quad$ **unspan**($op_{\Pi(C')}(\mathbf{Agg}_{\Pi(C')}(\Phi(C')))$) $\Leftarrow$ $op_{\Pi(C)}$(**unspan**($\mathbf{Agg}_{\Pi(C')}(\Phi(C'))$))
end

Data Transform Template - Unspan.

When an arc (and thus a compatibility map) exists from a node in the frontier to a node in the core, then unspan is used in the definition to map the data representation of an aggregate consisting of the nodes of the core and the frontier into an aggregate consisting of the nodes of the core. (We call it unspan, since it spans in the opposite direction of how we are building the aggregate, that is, from core to core and frontier.) The unspan function uses the compatibility maps to define how the data representations of the nodes of the core are generated from a data representation of a node in the frontier, using the notation $\{ c_1 \mid c_2 \}$ to denote that a value is computed in more than one way. We use $B[x \backslash exp]$ to mean, replace all occurrences of x in B with *exp*. The notation needs to be supplemented in this manner because multiple ways to compute a value must be maintained to ensure consistency among the components.

A number of simplifying assumptions have been made for this presentation. Only one component in the frontier is merged at a time. This result can be generalized to merge a collection of nodes in the frontier with the core. The definition in Figure 3.4 uses this approach where span_c coalesces $\texttt{Buf}_2$ and $\texttt{Buf}_3$ together with $\texttt{Buf}_1$. This algorithm treats the aggregate data structure as the product of the components, maintaining the component abstractions. In Figure 3.4 the abstractions are lifted; the data structure is the product of the fields of the components.

A Development of the Aggregate Definition

In order to show that the aggregate definition satisfies the aggregate specification, we start with the specification and use a constructive approach in developing the aggregate definition. The running example consists of three arbitrary components (which "represent" the same object), and we consider the case where the operation is defined in the second component. Three components are enough to consider all of the various integration possibilities. The aggregate specification is defined using axioms in Figure 6.4.

The first three axioms declare that there are consistency relations among the components. The next axiom defines the behavior of the operation on the aggregate datatype in terms of the second component. The remaining three axioms ensure that after applying the operation, all of the components remain consistent. There is such a set of axioms for each operation defined; for illustrative purposes only one set is shown.

$proj_1 : \quad agg \rightarrow c_1$
$proj_2 : \quad agg \rightarrow c_2$
$proj_3 : \quad agg \rightarrow c_3$

$map_2^1 : \quad c_2 \times c_1 \rightarrow bool$
$map_3^2 : \quad c_3 \times c_2 \rightarrow bool$
$map_3^1 : \quad c_3 \times c_1 \rightarrow bool$

axiom $proj_2(c)\ \ map_2^1\ \ proj_1(c)$
axiom $proj_3(c)\ \ map_3^2\ \ proj_2(c)$
axiom $proj_3(c)\ \ map_3^1\ \ proj_1(c)$

axiom $proj_2(op(agg)) = op_2(proj_2(agg))$

axiom $proj_2(agg)\ \ map_2^1\ \ proj_1(agg) \Longrightarrow proj_2(op(agg))\ \ map_2^1\ \ proj_1(op(agg))$
axiom $proj_3(agg)\ \ map_3^2\ \ proj_2(agg) \Longrightarrow proj_3(op(agg))\ \ map_3^2\ \ proj_2(op(agg))$
axiom $proj_3(agg)\ \ map_3^1\ \ proj_1(agg) \Longrightarrow proj_3(op(agg))\ \ map_3^1\ \ proj_1(op(agg))$

Figure 6.4: Aggregate Specification

For any aggregate definition of the datatype that we define, we must ensure that it satisfies these axioms (that comprise the aggregate specification). We start with a very simple definition (Figure 6.5) that assumes we have functional mappings in either direction between any two components. Then we generalize the result a little more to make it easier for the designer to define the datatype. In Figure 6.6 we relax the restriction that requires compatibility maps in either direction between any two components, to requiring a single compatibility map in one direction. In Figures 6.7, 6.8, and 6.9 we relax the restriction that requires any two components to be directly connected to simply requiring a connection, possibly through some number of intermediate components. We will see that care must be taken when dealing with many-to-one compatibility maps. Finally we produce a definition (seen in Figure 6.10) that allows us to exploit the interdependencies among the components for optimization purposes. This is the definition that is used in the algorithm.

We take a constructive approach by showing how to transform the axioms into the definition. Each section introduces a definition. The proof details on how it satisfies each of the axioms are contained in Appendix E.

Product of the Representations. We start off, for the sake of expediency, by defining the representation of the data aggregate as the product of the component representations,

type agg = Agg **of** $c_1 \times c_2 \times c_3$

and the operations as:

$$op(agg) \Leftarrow proj_j^{-1}(op_j(proj_j(agg)))$$

This definition is dependent on the ability to define the consistency relations as a pair of functions mapping from one representation to another and vice versa. The consistency

Given: op_2, $map_{1\to 2}$, $map_{2\to 1}$, $map_{1\to 3}$, $map_{3\to 1}$, $map_{2\to 3}$, $map_{3\to 2}$

local

$proj_2(Agg(c_1, c_2, c_3)) \Leftarrow c_2$

$proj_2^{-1}(x) \Leftarrow Agg(map_{2\to 1}(x), x, map_{2\to 3}(x))$

in

$op(agg) \Leftarrow proj_2^{-1}(op_2(proj_2(agg)))$

end

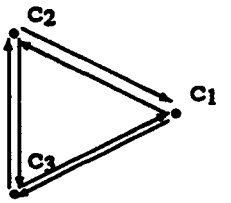


Figure 6.5: A Simple Definition

relation $x\ map_i^j\ y$ is defined using a pair of compatibility maps as $x = map_{j\to i}(y)$ and $y = map_{i\to j}(x)$. This requires that each compatibility map has an inverse since the aggregate must be able to be generated from any component. Then, taking the projection is simply extracting the appropriate field, for example, $proj_2(Agg(c_1, c_2, c_3)) = c_2$. Taking the inverse projection is simple as well, since the aggregate is easily generated from any component since there are mappings defined from any component to all the others.

We start with the axiom defining the behavior of the aggregate operation and apply transformations to obtain a functional definition for op.

$$proj_2(op(agg)) = op_2(proj_2(agg))$$

We take the inverse projection of each side, (we must show that $proj_2^{-1}$ is injective)

$$proj_2^{-1}(proj_2(op(agg))) = proj_2^{-1}(op_2(proj_2(agg)))$$

and then simplify (we must show that $proj_2^{-1}$ is a left inverse of $proj_2$).

$$op(agg)) = proj_2^{-1}(op_2(proj_2(agg)))$$

Grouping all the definitions together we get the definition shown in Figure 6.5. The components and the compatibility maps that are given are depicted to the right of the definitions (in the figure). Components are depicted as nodes labeled with the name of the component. Compatibility maps are depicted as directed arcs.

Reimplementing the Operations. We would like to relax the restriction that requires compatibility maps in both directions between any two components, to merely requiring a single compatibility map in one direction. There are two cases to consider, for either function that is removed. The consistency relation $x\ map_i^j\ y$ is defined using one or the other compatibility map as $x = map_{j\to i}(y)$ or $y = map_{i\to j}(x)$. Recall that it is possible to define a new implementation of an operation given a compatibility map. When the function translates from the old representation to the new we call it "span" (since it spans representations). We have the following definition for op′:

$$op'(span(x)) \Leftarrow span(op(x))$$

Given: op_2, $map_{2\to 1}$, $map_{3\to 2}$

local

$\underline{op_1}(map_{2\to 1}(c_2)) \Leftarrow map_{2\to 1}(op_2(c_2))$

$map_{3\to 2}(\underline{op_3}(c_3)) \Leftarrow op_2(map_{3\to 2}(c_3))$

in

$op(Agg(c_1, c_2, c_3)) \Leftarrow$

$Agg(c_1', c_2', c_3')$

where $c_1' = op_1(c_1)$

and $c_2' = op_2(c_2)$

and $c_3' = op_3(c_3)$

end

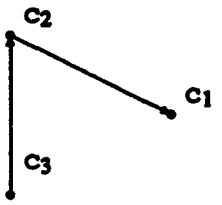


Figure 6.6: Product of the Operations

When the function translates from the new representation to the old, we call it "unspan." We have the following definition for op′:

$$unspan(\underline{op'}(x)) \Leftarrow op(unspan(x))$$

One way to implement the aggregate is to use as a representation an n-tuple of the representations of the various components. Then, the operations are defined over this n-tuple. Consider, for example, a component c_j in which the operation op_i is defined. We develop corresponding implementations of this operation for the other components by using data transform procedures.

local

$op_k(map_{j\to k}(c_j)) \Leftarrow map_{j\to k}(op_i(c_j))$

in

$op(Agg(\ldots, c_j, c_k, \ldots)) \Leftarrow Agg(\ldots, c_j', c_k', \ldots)$

where $c_j' = op_i(c_j)$

and $c_k' = op_k(c_k)$

end

Here, data transform procedures allow us to define op_k as an alternative implementation of op_i on the data representation of c_k. To do this, we depend on a compatibility map, $map_{j\to k}$, from c_j's representation into c_k's representation. If we had the inverse instead, we use the other form of the data transform procedure. The compatibility map, $map_{j\to k}$, respects the consistency relation, map_j^k.

Using this approach in our example, if we are given the operation op_2, and compatibility maps between the second component and each of the other components, we derive new implementations of the operation for the other components using the data transform definitions. Then we define the operation on the aggregate in terms of the component operations, where each component operation updates the appropriate subcomponents of the aggregate (see Figure 6.6).

The definition for op_1 demonstrates the use of a span function since $map_{2\to 1}$ translates from the old representation to the new representation in this case. The definition for

op_3 demonstrates the use of an unspan function since $map_{3\to 2}$ translates from the new representation to the old representation in this case. Unlike the previous definition, where the defining component was translated into each of the other components, here we are actually defining operations for each of the other components. If the appropriate compatibility map is available, then we have a choice between translation and deriving a new operation. To compute the new value of c_1, for example, we know how to translate between c_2 and c_1 and can define c_1' as $map_{2\to 1}(op_2(c_2))$. Alternately, we can derive a new operation, op_1, to compute the new value of c_1' using $op_1(c_1)$. Since there is no compatibility map from c_2 to c_3 we have no choice, but must derive a new function op_3 to compute the value for c_3'.

In this simplified presentation, we cannot get the result shown in the buffer example, where the cursor in $\texttt{Buf}_1$ is computed in terms of the sequences of lines in $\texttt{Buf}_3$, since components cannot interact. We see later how to obtain this result (seen in Figure 6.10). But first we deal with the restriction that this definition is only applicable when all the components are directly connected. We would like a definition that is also applicable when components are indirectly connected, through some number of intermediate components.

Showing Transitivity. We would like to relax the restriction that requires any two components to be directly connected by a compatibility map, to simply requiring a connection, possibly through some number of intermediate components. We must be careful to exclude intermediates that lose information. For example, the translation from a component that represented a buffer as text into a component that represented the buffer as s-expressions loses information about whitespace and newline positions. These were automatically excluded in the previous definitions because we could not write the required compatibility maps. Now that we are not required to write mappings in all cases, we must be careful. We can allow many-to-one mappings on the fringe of the component graph, but not within it where they might act as an intermediate.

Say we are given a direct compatibility map between c_3 and c_2 which we use to define op_3, an alternative implementation of op_2 for component c_3. We would like to replace this compatibility map by a compatibility map between c_3 and some intermediate, say, c_1, and a compatibility map between this intermediate and c_2 in the context of defining op_3. There are four cases to consider depending on which way the compatibility maps are defined. These cases can be iterated for arbitrary path lengths. In all cases, we derive the new operation, op_3, by first obtaining an intermediate operation, op_1, for the intermediate component.

1. Given the translations $g : c_2 \to c_1$ and $h : c_1 \to c_3$, we define the relation between c_3 and c_2 as: $h(g(b)) = c$, and define op_3 as:

$$\begin{aligned} \underline{op_1}(g(c_2)) &\Leftarrow g(op_2(c_2)) \\ \underline{op_2}(h(c_1)) &\Leftarrow h(op_1(c_1)). \end{aligned}$$

   Recall the example from Figure 6.6. If we did not have the direct connection between c_3 and c_2, $map_{3\to 2}$, but rather $map_{1\to 3}$, then op_3 must be defined indirectly in terms of op_1, which in turn must be related back to op_2 where the operation is originally defined (see Figure 6.7).

Given: op_2, $map_{2\to 1}$, $map_{1\to 3}$

local

$\underline{op_1}(map_{2\to 1}(c_2)) \Leftarrow map_{2\to 1}(op_2(c_2))$
$\underline{op_3}(map_{1\to 3}(c_1)) \Leftarrow map_{1\to 3}(op_1(c_1))$

in

$op(Agg(c_1, c_2, c_3)) \Leftarrow$
  $Agg(c_1', c_2', c_3')$
    where $c_1' = op_1(c_1)$
      and $c_2' = op_2(c_2)$
      and $c_3' = op_3(c_3)$

end

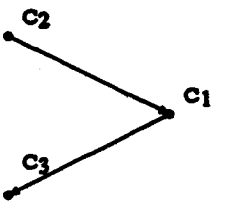


Figure 6.7: Transitivity — Case 1

2. Given the translations $g : c_1 \to c_2$ and $h : c_3 \to c_1$, we define the relation between c_3 and c_2 as: $g(h(c)) = b$, and define op_3 as:

$$g(\underline{op_1}(c_1)) \Leftarrow op_2(g(c_1))$$
$$h(\underline{op_3}(c_3)) \Leftarrow op_1(h(c_3)).$$

   This is similar to Case 1.

3. Given the translations $g : c_2 \to c_1$ and $h : c_3 \to c_1$, we define the relation between c_3 and c_2 as: $g(c_2) = h(c_3)$, and define op_3 as:

$$\underline{op_1}(g(c_2)) \Leftarrow g(op_2(c_2))$$
$$h(\underline{op_3}(c_3)) \Leftarrow op_1(h(c_3)).$$

   Recall the example from Figure 6.6. If we did not have $map_{3\to 2}$ but rather $map_{3\to 1}$, then op_3 must be defined in terms of op_1, which in turn must be related back to op_2 where the operation is originally defined (see Figure 6.8).

   Our system must remain consistent, so we consider the possibility when the intermediate component loses information. In this case our equation does not express our intention of consistency. Take for example, the s-expression component. We easily define compatibility maps from $\texttt{Buf}_1$ to $\texttt{Buf}_s$ and from $\texttt{Buf}_2$ to $\texttt{Buf}_s$. Each compatibility map loses information about whitespace. We cannot combine these two compatibility maps to obtain one that translates between $\texttt{Buf}_1$ and $\texttt{Buf}_2$. We must add the constraint that the compatibility maps be injective. (We do not have to introduce this constraint to the other cases because we are not able to define the required compatibility maps.)

4. Given the translations $g : c_1 \to c_2$ and $h : c_1 \to c_3$, we define the relation between c_3 and c_2 as: $\exists x : c_1 . g(x) = c_2$ and $h(x) = c_3$, and define op_3 as:

$$g(\underline{op_1}(c_1)) \Leftarrow op_2(g(c_1))$$
$$\underline{op_3}(h(c_1)) \Leftarrow h(op_1(c_1)).$$

```
Given: op2, map2→1, map3→1
local
   op1(map2→1(c2))   ⇐   map2→1(op2(c2))
   map3→1(op3(c3))   ⇐   op1(map3→1(c3))
in
   op(Agg(c1, c2, c3))   ⇐
      Agg(c1', c2', c3')
         where c1'   =   op1(c1)
            and c2'  =   op2(c2)
            and c3'  =   op3(c3)
end
```

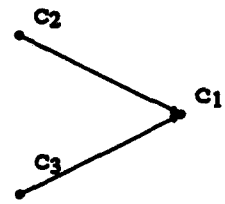


Figure 6.8: Transitivity — Case 3

```
Given: op2, map1→2, map1→3
local
   map1→2(op1(c1))   ⇐   op2(map1→2(c1))
   op3(map1→3(c1))   ⇐   map1→3(op1(c1))
in
   op(Agg(c1, c2, c3))   ⇐
      Agg(c1', c2', c3')
         where c1'   =   op1(c1)
            and c2'  =   op2(c2)
            and c3'  =   op3(c3)
end
```

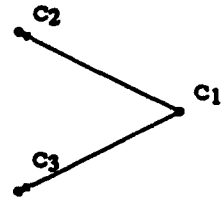


Figure 6.9: Transitivity — Case 4

Here we have a different set of mapping functions, but again op_3 must be defined in terms of op_1, which in turn must be related back to op_2 where the operation is originally defined. This results in a new definition of op (see Figure 6.9).

Incrementally Building the Aggregate. A more flexible approach is to arrange for all of the component representations to be available for the operation definitions in order to take advantage of any interrelationships among the various representations. Then we eliminate the restrictions seen in Figure 6.6 where component fields that make up the aggregate representation could not interact. Going back to this previous aggregate definition, instead of actually deriving implementations for the component operations, represented by op_k, the data transform procedure itself is symbolically manipulated to yield a new definition for the aggregate.

```
local
   span(c_j)  ⇐  Agg(..., c_j, c_k, ...)
                    where ... c_k  =  map_{j→k}(c_j) ...
in
   op(span(c_j))  ⇐  span(op_i(c_j))
end
```

This is accomplished by, in effect, unfolding the component operation definitions within the body of the aggregate operation. The abstraction boundaries of the components are lifted to facilitate improvements in the efficiency of the aggregate data representation and operations. The span function is abstracted from the definition for notational convenience, and to put the definition in the proper form for manipulation as a data transform procedure. If we had the inverse compatibility map available, then the unspan form of the data transform procedure is used instead.

We think of this process operationally as incrementally building the aggregate, repeatedly merging adjacent components until the single aggregate is left. The definitions of the operations and spanning functions are obtained mechanically by considering the order in which the components must be "merged." The basic idea is to consider the buffer definition as a graph, where the components are nodes and the compatibility maps are directed arcs. The operations for each component must be reimplemented to operate on the new aggregate representation. This is done in stages. Starting at the node representing the component where the operations are defined, all connected nodes are merged into a new "coalesced" node using a variant to the data transformation techniques. This coalescing of connected nodes is repeated until the graph collapses into a single node.

Returning to our example (see Figure 6.10), we start with the given definition for op_2. We merge the second component (where the operation is defined) with the adjacent first component to obtain an intermediate operation $op_{1\times2}$. Then we merge this intermediate aggregate with the now adjacent third component to obtain the operation op on the aggregate data structure. Notice that span and unspan are being introduced explicitly for the first time. In the previous sections, the compatibility maps served implicitly in that capacity. However, the expressions here are more complex so that span and unspan functions must be introduced in order to get the expressions into a form that we know how to manipulate.

These are precisely the span and unspan definitions used in the algorithm presented at the beginning of this section. They satisfy the properties given in Definition 9. By construction, the span function is guaranteed to be injective. Since the component argument appears in the aggregate result, then two distinct components always yield two distinct aggregates. Also by construction, the unspan function is guaranteed to be surjective. Since the fields in the aggregate result are a subset of the fields in the aggregate argument, then every element of the result is in the image of unspan.

6.1.5 Aggregate Implementation

Continuing the implementation of the specification *P*, the data transform procedures from the aggregate definition are transformed into functional definitions to yield an aggregate prototype.

```
Given: op2, map2→1, map3→1
local
    span(c2)  ⇐
        Aggs(c1, c2) where c1  =  map2→1(c2)
    unspan(Agg(c1, c2, c3))  ⇐
        Aggs({ c1 | map3→1(c3) }, c2)
in
    op1×2(span(c2))  ⇐
        span(op2(c2))
    unspan(op(Agg(c1, c2, c3)))  ⇐
        op1×2(unspan(Agg(c1, c2, c3)))
end
```

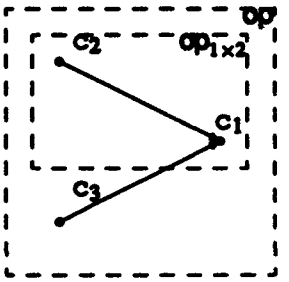


Figure 6.10: Incremental Merging

Definition 10. An aggregate *prototype* is a refinement of the aggregate definition where the data transform procedures have been transformed into functional definitions to produce the first executable system.

We saw an example of a prototype buffer in Figure 3.5.

6.1.6 Optimization

The functional definitions of the aggregate prototype are further refined for optimization purposes to yield an aggregate implementation.

Definition 11. An aggregate *implementation* is a refinement of the prototype providing an "efficient" implementation of the datatype.

Efficiency is measured in terms of the developers needs, and involves, for example, tradeoffs between the amount of space the program uses and the time required to run the program. We saw an example of an optimized buffer in Figure 3.6.

6.2 Module Transformation Rules

This section provides a more rigorous explanation of the module transformation rules: *translate*, *shift*, *expose*, *incorporate*, and *release*. They were initially presented in Section 2.3; the reader may want to refer back to that section to review the notation and naming conventions. Each subsection includes a description of the module transformation rule and the steps in applying the transformation. Strictly speaking, the *transformation* is a single step. The surrounding steps to put the program into the proper form is the transformation *strategy*.

6.2.1 Translate

The *translate* transformation is used to change the representation of a datatype and/or move computation along the data paths of a program. This change in representations is expressed by a function that maps from the original representation into the new one. The transformation provides a mechanical means to reimplement the operations of this datatype on the alternative data representation. The meaning of the abstract datatype, however, remains the same. This transformation differs from *shift* [76] in its use in the integration of components; *shift* is used to optimize within a single component, *translate* is used to integrate between components. Rather than "synthesizing" the function within a component that is used to transform the original program into a more efficient one, *translate* uses an "analytic" approach. The function is introduced at the system level between two distinct components in order to integrate them. In addition, when *shift* requires an inverse translation function that is difficult to define, *translate* could be used as an alternative. With the emphasis on integrating datatypes, this transformation differs from work done by Darlington [17, 18] on synthesizing implementations from algebraic specifications. Harrison and Khoshnevison [43] have developed an automated system for implementing datatypes for a limited language where they synthesize the inverse mapping function.

In order to understand the *translate* transformation, we examine a representative selection of operations for the datatype, that produce an instance of the type, operate on the type, and reveal some information about the type. (Of course there may be additional parameters besides the type of interest.)

```
Signature DTYPE = sig
    type dtype
        gen : dtype
        ext : dtype → dtype
        obs : dtype → ν
    end
```

Given a spanning function:

$U : \alpha' \rightarrow \alpha$
unspan : Dtype'.dtype → Dtype.dtype
unspan(Dtype'.Abs(a)) ⇐ Dtype.Abs($U(a)$)

Replace:

```
Structure Dtype : DTYPE = struct
    type dtype = Abs of α
        gen ⇐ Abs(G)
        ext(Abs(a)) ⇐ Abs(E(a))
        obs(Abs(a)) ⇐ O(a)
    end
```

By:

```
Structure Dtype' : DTYPE = struct
    type dtype = Abs of α'
        unspan(gen)    ⇐  Dtype.gen
        unspan(ext(a))  ⇐  Dtype.ext(unspan(a))
        obs(a)    ⇐  Dtype.obs(unspan(a))
    end
```

Using the notion of correctness relation [46, 78], we can ensure that unspan is a valid abstraction function. An abstraction function is a surjective strong-homomorphism from the representation, B, to the abstraction, A. We construct an abstraction function,

$$h_{\texttt{dtype}} : \texttt{Dtype}'_{\texttt{dtype}} \rightarrow \texttt{Dtype}_{\texttt{dtype}} \qquad (i.e.,\ h_{\texttt{dtype}} : \alpha' \rightarrow \alpha).$$

We impose the requirement that it be surjective. (A requirement that must be ensured by the software designer.) Then we can show that h is a strong partial homomorphism, considering each operator in turn.

$$h_{\texttt{dtype}}(B_{\texttt{gen}}) = A_{\texttt{gen}}$$
$$h_{\texttt{dtype}}(B_{\texttt{ext}}(x)) = A_{\texttt{ext}}(h_{\texttt{dtype}}(x))$$
$$B_{\texttt{obs}}(x) = A_{\texttt{obs}}(h_{\texttt{dtype}}(x))$$

In our transformation, where we treat `Dtype'` as B and `Dtype` as A, these equations are satisfied by constructor, $h_{\texttt{dtype}}$ = unspan. A similar argument holds when we have span (a representation function) instead.

Applying the transformation. This transformation is the important step in a series of steps that produce the new implementation of the datatype. The data representation and the implementation of the operations change, but the meaning of the datatype remains the same. The high-level steps of the transformation are: (1) define the new implementations as data transform procedures; (2) unfold all old definitions; (3) "bridge" the old and new representations; and (4) fold the spanning function. Preliminary work was done in an Ergo Seminar on Inferential Programming by Elliott [24].

The operations are defined on the datatype with representation α. For example, obs takes an instance of the datatype as an argument, reveals the underlying data representation of the abstraction (represented by a) and returns the result of performing some operation on it (represented by the pattern O).

```
Structure Dtype : DTYPE = struct
    type dtype = Abs of α
        gen    ⇐  Abs(G)
        ext(Abs(a))    ⇐  Abs(E(a))
        obs(Abs(a))    ⇐  O(a)
    end
```

The function span maps the given datatype representation into the new representation. From now on, `Dtype`.Abs and `Dtype'`.Abs are abbreviated as Abs and Abs'.

$$\text{span}(\text{Abs}(a)) \Leftarrow \text{Abs}'(S(a))$$

The new implementations of the operations are defined in terms of the original implementations on α (as data transform procedures). Strictly speaking, this is the *translate* transformation.

```
Structure Dtype' : DTYPE = struct
    type dtype = Abs of α'
        gen ⇐ span(Dtype.gen)
        ext(span(Abs(a))) ⇐ span(Dtype.ext(A))
        obs(span(Abs(a))) ⇐ Dtype.obs(A)
        span(Abs(a)) ⇐ Abs'(S(a))
    end
```

The old operation definitions are mechanically "unfolded," that is, the names of the operations are replaced by their bodies.

```
Structure Dtype' : DTYPE = struct
    type dtype = Abs of α'
        gen ⇐ span(Abs(G))
        ext(span(Abs(a))) ⇐ span(Abs(E(a)))
        obs(span(Abs(a))) ⇐ O(a)
        span(Abs(a)) ⇐ Abs'(S(a))
    end
```

The span function on the righthand side is likewise mechanically "unfolded."

```
Structure Dtype' : DTYPE = struct
    type dtype = Abs of α'
        gen ⇐ Abs'(S(G))
        ext(span(Abs(a))) ⇐ Abs'(S(E(a)))
        obs(span(Abs(a))) ⇐ O(a)
        span(Abs(a)) ⇐ Abs'(S(a))
    end
```

If the inverse of span could be obtained, then deriving new implementations of the operations is easier. Simply map the new datatype into the old representation, perform the operation, and then map the datatype back into the new representation. Obtaining the inverse may not be practical since the spanning function may not always be one-to-one, and, even if it were, there may be no easy way to obtain it. Rather than coming up with the inverse explicitly, it is sometimes possible to use syntactic manipulations and simplifications to in effect, "invert" the spanning function. This is accomplished by simplifying the expressions to match the new representation expressed in the spanning function. This is where insights about the domain from the developer are needed.

```
Structure Dtype' : DTYPE = struct
    type dtype = Abs of α'
        gen  ⇐  Abs'(G')
        ext(span(Abs(a)))  ≐  Abs'(E'(S(a)))
        obs(span(Abs(a)))  ⇐  O'(S(a))
        span(Abs(a))  ⇐  Abs'(S(a))
    end
```

Now that the expressions in the operations match the spanning function, we mechanically "fold" span on the righthand side.

```
Structure Dtype' : DTYPE = struct
    type dtype = Abs of α'
        gen  ⇐  Abs'(G')
        ext(span(Abs(a)))  ⇐  Abs'(E'(Rep'(span(Abs(a)))))
        obs(span(Abs(a)))  ⇐  O'(Rep'(span(Abs(a))))
        span(Abs(a))  ⇐  Abs'(S(a))
    end
```

Since all instances of the datatype appear in the context of span(Abs(*a*)) which is the new datatype, it is renamed.

```
Structure Dtype' : DTYPE = struct
    type dtype = Abs of α'
        gen  ⇐  Abs'(G')
        ext(Abs'(a))  ⇐  Abs'(E'(a))
        obs(Abs'(a))  ⇐  O'(a)
    end
```

New implementations of the operations (represented by G', E', and O') have been derived that operate on the new data structure, α', directly.

6.2.2 Shift

The *shift* transformation is used to move computation along the data paths of a program to increase the efficiency of the program (*eg.*, moving computation on a data structure from when it is accessed to when it is generated). This may change the data representation of a datatype but the meaning of the abstract datatype remains the same. The idea of a *shift* transformation was presented by Jørring and Scherlis [49, 76].

```
Signature DTYPE = sig
    type dtype
        gen : dtype
        obs : dtype → ν
    end
```

Replace:

```
Structure Dtype : DTYPE = struct
   type dtype = Abs of α
      gen  ⇐  Abs(G)
      obs(Abs(a))  ⇐  Rep'(span(Abs(a)))
      span(Dtype.Abs(a))  ⇐  Dtype'.Abs(O(a))
   end
```

By:

```
Structure Dtype' : DTYPE = struct
   type dtype = Abs of α'
      gen  ⇐  span(Abs(G))
      obs(Abs(a))  ⇐  Rep'(Abs(a))
      span(Dtype.Abs(a))  ⇐  Dtype'.Abs(O(a))
   end
```

Shift is a special case of *translate*. We demonstrate this by using *translate* to effect the shift from the original definition of the operations in `Dtype` to the new definitions in `Dtype'`.

```
gen  ⇐  Abs(G)
obs(Abs(a))  ⇐  Rep'(span(Abs(a)))
```

Jørring and Scherlis perform the actual shift by replacing Abs with span ∘ Abs, and span by the identity function. Instead, we introduce a step for defining new definitions for the operations using the *translate* transformation to define alternative implementations.

```
gen  ⇐  span(Dtype.gen)
obs(span(Abs(a)))  ⇐  Dtype.obs(Abs(a))
```

Then we mechanically "unfold" the definitions of the operations on the righthand side. This puts the generator in the desired format, (*i.e.*, replacing Abs by span ∘ Abs). In the observer function, the datatype is already in the context of span on the righthand side. This matches the occurrence on the lefthand side introduced in the definition.

```
gen  ⇐  span(Abs(G))
obs(span(Abs(a)))  ⇐  Rep'(span(Abs(a)))
```

Since the datatype in obs appears only in the context of span it is renamed. This produces the same result as Scherlis obtains in replacing span by the identity function.

```
gen  ⇐  span(Abs(G))
obs(Abs(a))  ⇐  Rep'(Abs(a))
```

We then simplify, removing the abstraction boundary in obs.

```
gen  ⇐  span(Abs(G))
obs(Abs(a))  ⇐  a
```

We now have the transformed program of `Dtype'` where span has been shifted from obs to gen. A similar argument holds when shifting from gen to obs.

Applying the transformation. The steps for transforming the operations follow for "advancing" computation from the observer obs to the generator gen. It is also possible to "delay" computation from a generator to an observer. The high-level steps of the transformation are: (1) introduce the intended new boundary; (2) abstract the code segment "between" the old representation and the new abstraction into a new function; (3) accomplish the actual shift; and (4) simplify.

We start with the original definition,

```
Structure Dtype : DTYPE = struct
  type dtype = Abs of α
    gen  ⇐  Abs(G)
    obs(Abs(a))  ⇐  O(a)
  end
```

and introduce an abstraction boundary in the observer function.

```
Structure Dtype : DTYPE = struct
  type dtype = Abs of α
    gen  ⇐  Abs(G)
    obs(Abs(a))  ⇐  Rep'(Abs'(O(a)))
  end
```

The next step is to advance computation by mechanically "folding" the new abstraction into a span function. The span function takes the original abstract datatype and produces a new abstraction.

```
Structure Dtype : DTYPE = struct
  type dtype = Abs of α
    gen  ⇐  Abs(G)
    obs(Abs(a))  ⇐  Rep'(span(Abs(a)))
    span(Abs(a))  ⇐  Abs'(O(a))
  end
```

The datatype is now in the correct format to apply the *shift* transformation.

```
Structure Dtype' : DTYPE = struct
  type dtype = Abs of β
    gen  ⇐  span(Abs(G))
    obs(Abs(a))  ⇐  a
    span(Abs(a))  ⇐  Abs'(O(a))
  end
```

We simplify by unfolding span in the generator.

Structure Dtype' : DTYPE = **struct**
  type dtype = Abs **of** β
    gen $\Leftarrow$ Abs'($O(G)$)
    obs(Abs'(a)) $\Leftarrow$ a
  end

We see that shifting under these conditions is a special case of the more general *translate* data transformation technique, where the new generator G' is $O(G)$, and the new observer O' is the identity function. Since the span function is uncovered from the observer function, it simply drops out as we shift the computation over to the generator function.

6.2.3 Expose

The *expose* transformation is used to reveal the underlying type structure. This has the effect of moving the boundary of the type "inward." The *expose* transformation was presented by Scherlis [76].

Given spanning functions:

$S : T \rightarrow T_1 \times \cdots \times T_n$
$U : T_1 \times \cdots \times T_n \rightarrow T$
$U = S^{-1}$

span(Dtype.Abs(a)) $\Leftarrow$ Dtype'.Abs($S(a)$)
unspan(Dtype'.Abs(a)) $\Leftarrow$ Dtype.Abs($U(a)$)

Replace:

Structure Dtype : DTYPE = **struct**
  type dtype = Abs **of** α
    gen $\Leftarrow$ Abs($U(G)$)
    ext(a) $\Leftarrow$ Abs($U(E(S(\text{Rep}(a))))$)
    obs(a) $\Leftarrow$ $O(S(\text{Rep}(a)))$
  end

By:

Structure Dtype' : DTYPE = **struct**
  type dtype = Abs **of** α
    gen $\Leftarrow$ unspan($\langle \text{Abs}_1, \ldots, \text{Abs}_n \rangle(G)$)
    ext(a) $\Leftarrow$ unspan($\langle \text{Abs}_1, \ldots, \text{Abs}_n \rangle(E(\langle \text{Rep}_1, \ldots, \text{Rep}_n \rangle(\text{span}(a))))$)
    obs(a) $\Leftarrow$ $O(\langle \text{Rep}_1, \ldots, \text{Rep}_n \rangle(\text{span}(a)))$
  end

The *expose* transformation replaces all instances of Abs $\circ$ U by unspan $\circ$ $\langle \text{Abs}_1, \ldots, \text{Abs}_n \rangle$ and all instances of $S \circ$ Rep by $\langle \text{Rep}_1, \ldots, \text{Rep}_n \rangle \circ$ span. The bodies of span and unspan are represented by S and U.

Expose can be thought of as a "strategy" composed of more basic steps. These steps are explained in terms of the simpler transformation steps, introducing an abstraction boundary and folding the definition for span or unspan. For example, starting with Abs $\circ$ U, we introduce an abstraction boundary to get Abs $\circ$ U $\circ$ $\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle$ $\circ$ $\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle$. But the first three composed operations is the definition of unspan, so folding obtains, unspan $\circ$ $\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle$.

Here are the steps for replacing all instances of Abs $\circ$ U by unspan $\circ$ $\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle$:

$\mathsf{Abs} \circ U$
$\mathsf{Abs} \circ U \circ \langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle \circ \langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle$
unspan $\circ$ $\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle$

Here are the steps for replacing all instances of S $\circ$ Rep by $\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle$ $\circ$ span:

$S \circ \mathsf{Rep}$
$\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle \circ \langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle \circ S \circ \mathsf{Rep}$
$\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle \circ$ span

Applying the transformation. The high-level steps of this transformation are: (1) manipulate the type so that all instances of Rep appear in the context S $\circ$ Rep and all instances of Abs appear in the context Abs $\circ$ U; (2) move the boundary of the type inward; and (3) excise the spanning functions.

We start with the original definition,

```
Structure Dtype : DTYPE = struct
    type dtype = Abs of α
        gen  ⇐  Abs(G)
        ext(a)  ⇐  Abs(E(Rep(a)))
        obs(a)  ⇐  O(Rep(a))
    end
```

and manipulate the representation of the type so that all instances of Rep appear in the context S $\circ$ Rep and all instances of Abs appear in the context Abs $\circ$ U. This is done using simplification steps, the fold transformation, and insight from the software developer.

$S : \alpha \rightarrow \alpha_1 \times \cdots \times \alpha_n$
$U : \alpha_1 \times \cdots \times \alpha_n \rightarrow \alpha$
$U = S^{-1}$

```
Structure Dtype : DTYPE = struct
    type dtype = Abs of α
        gen  ⇐  Abs(U(G'))
        ext(a)  ⇐  Abs(U(E'(S(Rep(a)))))
        obs(a)  ⇐  O'(S(Rep(a)))
    end
```

The datatype is now in the correct format to apply the *expose* transformation; all instances of $S \circ \mathsf{Rep}$ are replaced by $\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle \circ \mathsf{span}$ and all instances of $\mathsf{Abs} \circ U$ are replaced by $\mathsf{unspan} \circ \langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle$.

Structure `Dtype` : `DTYPE` = **struct**

type `dtype` = Abs **of** α

gen $\Leftarrow$ unspan($\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle(G')$)
ext(a) $\Leftarrow$ unspan($\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle(E'(\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle(\mathsf{span}(a)))))$
obs(a) $\Leftarrow$ $O'(\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle(\mathsf{span}(a)))$

span(a) $\Leftarrow$ $\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle(S(\mathsf{Rep}(a)))$
unspan(a) $\Leftarrow$ $\mathsf{Abs}(U(\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle(a)))$

end

This has the effect of moving the boundary of the type "inward." In Scherlis' paper, the next step is to use the *release* transformation to to excise the span and unspan portions from the type. Here, we instead make use of the *translate* transformation to derive a new implementation for the type, revealing the underlying data structure, the tuple $(\alpha_1 \times \ldots \times \alpha_n)$.

Structure `Dtype'` : `DTYPE` = **struct**

type `dtype` = Abs **of** $(\alpha_1 \times \ldots \times \alpha_n)$

gen $\Leftarrow$ span(`Dtype.gen`)
ext(a) $\Leftarrow$ span(`Dtype.ext`(unspan(a)))
obs(a) $\Leftarrow$ `Dtype.obs`(unspan(a))

span(a) $\Leftarrow$ $\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle(S(\mathsf{Rep}(a)))$
unspan(a) $\Leftarrow$ $\mathsf{Abs}(U(\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle(a)))$

end

Next we mechanically "unfold" the old operation definitions. The collection of abstraction functions, $\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle$, is applied to an n-tuple to create an n-tuple of abstractions. The collection of representation functions is similarly defined.

Structure `Dtype'` : `DTYPE` = **struct**

type `dtype` = Abs **of** $(\alpha_1 \times \ldots \times \alpha_n)$

gen $\Leftarrow$ span(unspan($\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle(G')$))
ext(a) $\Leftarrow$ span(unspan($\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle(E'(\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle(\mathsf{span}(\mathsf{unspan}(a)))))))$
obs(a) $\Leftarrow$ $O'(\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle(\mathsf{span}(\mathsf{unspan}(a))))$

span(a) $\Leftarrow$ $\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle(S(\mathsf{Rep}(a)))$
unspan(a) $\Leftarrow$ $\mathsf{Abs}(U(\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle(a)))$

end

Simplify, this is easy since span and unspan cancel out.

Structure `Dtype'` : `DTYPE` = **struct**

type `dtype` = Abs **of** $(\alpha_1 \times \ldots \times \alpha_n)$

gen $\Leftarrow$ $\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle(G')$
ext(a) $\Leftarrow$ $\langle \mathsf{Abs}_1, \ldots, \mathsf{Abs}_n \rangle(E'(\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle(a)))$
obs(a) $\Leftarrow$ $O'(\langle \mathsf{Rep}_1, \ldots, \mathsf{Rep}_n \rangle(a))$

end

6.2.4 Incorporate

The *incorporate* transformation moves an external function (or module) into the public part of a type, or a public function (or module) into the private part of a type. Declarations in the incorporated structure must be evaluated in the defining environment and name clashes avoided (cf. Standard ML semantics of the "open" statement). There must be no external references when moving a public structure into the private part of the type. This transformation is used for specializing modules in the context they appear in by moving external functions or subcomponents into a module. The *incorporate* transformation was presented by Scherlis [76]. Similar ideas were presented by Wile [91] based on Clear [13]. Similar applications to modules are found in the work on parameterization in OBJ [33] and in Tracz's thesis [89] where *incorporate* is an instance of "removal of horizontal structure — inheritance hierarchy flattening." Any module that imports another module can be reduced to a single module with the same functionality.

Since we do not need to access the internal structure of the datatype, we use a general function f (where $\{\_\}^*$ denotes multiple instances) rather than gen, ext, and obs.

Replace:

Signature DTYPE = **sig**
  $\{f : t\}^*$
  end
Structure Dtype : DTYPE = **struct**
  $\{f(v^*) \Leftarrow b\}^*$
  end
$g(v^*) : s \Leftarrow e$

By:

Signature DTYPE = **sig**
  $g : s$
  $\{f : t\}^*$
  end
Structure Dtype : DTYPE = **struct**
  $g(u^*) \Leftarrow e$
  $\{f(v^*) \Leftarrow b\}^*$
  end

This is a one step process, so unlike the other transformations, there is no section for applying the transformation.

6.2.5 Release

The *release* transformation moves a function (or module) out of the public or private part of a type, (provided the function definition does not contain any references to the abstraction functions for the type). Naming conflicts must also be handled (*eg.*, by renaming). The *release* transformation was presented by Scherlis [76].

Replace:

```
Signature DTYPE = sig
    g : s
    {f : t}*
    end
Structure Dtype : DTYPE = struct
    g(u*)  ⇐  e
    {f(v*)  ⇐  b}*
    end
```

By:

```
Signature DTYPE = sig
    {f : t}*
    end
Structure Dtype : DTYPE = struct
    {f(v*)  ⇐  b}*
    end
g(v*) : s  ⇐  e
```

This is a one step process, so unlike the other transformations, there is no section for applying the transformation.

6.3 Limitations and Benefits of the Semantic Model

The framework for the module transformation system described in this chapter uses: (1) an algebraic model to explain the meaning of integration; (2) a notion of a correctness relation to explain the meaning of transformations on data representations; and (3) theories to explain the meaning of transformations on abstract interfaces. The motivation for developing a framework is to enhance the understanding of the terms used informally, to provide structure for aiding the software developer in using this approach, and to provide insight into automating the process. Therefore the emphasis of this chapter has been on developing enough of a framework to understand the process rather than developing all the details of the semantic model.

The dependency of using Standard ML in this framework is on the module system, rather than on the language itself. Required extensions to the notation to express the transformation process include: axioms [72], views [33], expression procedures [74], and a means to express alternative solutions. Axioms are needed to enrich the expressive power of Standard ML so that the properties of the aggregate specification can be defined. Views enable the software designer to express how the components are related through consistency relations. Expression procedures provide the notation needed to express the initial definition of the aggregate upon which module transformation rules can be applied. Alternative solutions are used to maintain the internal consistency of the aggregate.

The benefits of this semantic model come from its simplicity and the insights it provides for automation. This semantic model could be enriched in a number of ways. The integration process described in Section 6.1 implements consistency relations in terms of translation functions. Higher-level abstractions of consistency that could provide a better model for explaining the design and manipulation of module interconnections are discussed in Section 8.2.4. An outline of the module transformation rules was provided in Section 6.2; this description would benefit from a richer model for interfaces, exports, imports, and a more abstract notion of equivalence (see Section 8.2.3). More details on how the module transformation system compares to related areas of work are discussed in Chapter 7.

Chapter 7

Related Work

Section 7.1 begins with an evaluation of the module interface transformation system (referred to as MTS in the discussion that follows) by comparing it to traditional approaches in software engineering to determine how well the approach addresses the problems of integrating module interfaces raised in the beginning of the thesis (Section 1.2). Then, the following four sections evaluate the utility of the MTS techniques by comparing them to several related areas of work: building systems from software components (Section 7.2); defining and managing representations (Section 7.3); data design and refinement (Section 7.4); and adapting interfaces (Section 7.5). Some of the related methods have similar motivations or shared techniques while others are complementary approaches. Data transformation techniques that this research builds upon were discussed in Section 2.2 and Section 6.2. Finally application domains for these techniques are discussed in Section 7.6.

7.1 Traditional Solutions — Revisited

The demonstrated MTS approach addresses the problems of integrating module interfaces (raised earlier in Section 1.2) in the subsections that follow. The areas of related work are grouped according to these four points which categorize when agreement on interfaces in the design must be reached, and is evaluated with respect to: scalability, expressiveness, appropriateness of representations, interface agreement, adaptability, correctness, performance, and automation. All of these criteria are not considered in each of the sections that follow; rather, the important points are raised as appropriate.

Building Systems from Software Components. Rather than requiring an *a priori* agreement on data representations, complex datatypes are defined as a collection of separate modules that are systematically merged using formal methods to derive the module interfaces and efficient representations.

When good data representations are difficult to design, especially when there is not much experience in the particular application domain, the system designer can define a complex datatype as a collection of separate modules. These modules are systematically merged using formal methods. The methods also facilitate adaptation, in line with evolutionary

models of development, since the process of adding a new component is similar to the process of merging the original components.

In Section 7.2, specification languages, module languages, and domain languages are evaluated with respect to expressiveness and scalability to determine how well they support building systems from collections of modules or software components.

Defining and Managing Representations. Rather than mediating representations through intermediate translation functions, new module interfaces are derived that interact directly.

This has been the primary focus of this research. Translation functions provide a notion of consistency among the components and transformations are applied to derive new module interfaces that interact directly. The requirements for translation functions between all components were relaxed to allow for indirect connections (via an intermediate component) and connections in one direction only.

In Section 7.3, techniques for defining and managing representations are evaluated with respect to appropriateness of representations, interface agreement, and adaptability.

Data Design and Refinement. Rather than leaving data design decisions to a compiler when using very-high-level languages, the software designer is involved in defining and organizing module interfaces.

The decision was made to involve the software designer by giving input to an interactive system in order to allow for rich abstractions for user-defined types. Once formalized it may be possible to more fully automate the approach; this has been left to future research. See, for example, an automation-based software development paradigm presented by Balzer [4]. Harrison and Khoshnevison [43] have an automated system for implementing datatypes; they use an approach similar to MTS in requiring the designer to specify an abstraction function (essentially a translation function), but use a restrictive language to automatically synthesize the inverse mapping function to define the implementation. Simplifications can then be applied to derive functions that operate on the new data structure directly.

In Section 7.4, frameworks and programming environments are evaluated with respect to correctness, performance, and automation, to determine how well they support the process of data design and refinement.

Adapting Interfaces. Rather than requiring *a priori* agreement on abstract interfaces, new types may be defined as extensions of existing ones (*eg.*, using object-oriented techniques such as inheritance) and new module interfaces derived.

The contribution in this area is in developing a complementary approach that builds on using modules and inheritance to adapt abstract interfaces and then applies derivations to specialize the implementation of the inherited module in the context of the new module. This approach has many similarities with mediating representations through translation functions and can make use of the experience learned from this earlier effort.

In Section 7.5, object-oriented techniques and transformation systems that support evolution are evaluated with respect to interface agreement and adaptability by means of modifying existing interfaces.

7.2 Building Systems from Software Components

In this section, specification languages, module languages, and domain languages are evaluated with respect to expressiveness and scalability, to determine how they support building systems from software components. Evaluating scalability raises the question, what techniques are available for managing larger programs? This is a major source of motivation for this investigation of transformations and module interfaces. Evaluating expressiveness raises the question, to what degree are the conceptual properties of the problem reflected in the syntax of the language? Module facilities enable more explicit representation of systems architecture and, through information hiding, enable components to be designed and developed separately. The challenge is to develop module mechanisms and formal methods approaches that can exploit modularity, and to develop formal methods approaches that support aggregation and integration of components for performance.

Even though the MTS approach is focused at the system design level, which requires a "programming language" in order to manipulate data representations, specification languages [92] give us insights into the structuring of large programs. Larch [38] and Z [84] are representative of recent developments in specification languages for specifying large programs.

The *trait* is the basic module of specification in Larch. A trait introduces operators and specifies their properties (via algebraic specifications). Sometimes the operators correspond to an abstract datatype; sometimes they do not since it may be desirable to specify properties that do not quite constitute a type. Traits can be combined to form richer traits, and a library of abstract interfaces has been constructed to aid the software designer. Components in MTS, like traits, describe a collection of operations that may not constitute a datatype.

The *schema* is the basic module of specification in Z. A schema introduces domains and operators and specifies their properties (via set theory). Schemas can be combined in various ways to build larger programs as defined by the schema calculus. For example, Sufrin [88] uses Z to develop a display editor. Z is a model-oriented specification language where one defines a system's behavior by constructing a model of the system in terms of mathematical structures such as tuples and sequences. Trying to develop a more extensive editor from this definition provided the motivation for introducing components that are views of some common datatype since it is difficult to create an initial model of the buffer that lends itself to specifying all of the various operations. Constraints in MTS, like Z invariants, were introduced to describe properties of the datatype that are not restricted to a particular operation.

Although these specification languages have similar goals in connecting software components, they are descriptive. They may describe what it means for components to be interrelated but not how to integrate them. This thesis is addressing the latter problem and is focused at the program design level where module interfaces and data representations are manipulated. Since specifications deal with a higher level of abstraction, it is necessary to use a programming language in which abstract interfaces and data representations can be defined and manipulated. However, these specification languages motivated adding components and constraints to the notation and influenced the style of the editor derivation.

From the many programming languages, Standard ML [61] was chosen because it has an elegant module facility and a fully defined semantics. Moreover, Extended ML [72] adds a useful extension to the language, the ability to add axioms, and gives a precise meaning to decomposing, refining, and composing programs. This gives the software designer a wide-spectrum language to represent higher-level specifications that can be refined to an implementable subset. Although the decision was made to base the notation on Standard ML, the transformation techniques are language independent and can be applied to other languages with modules such as Ada, Clu, and Modula 3.

Goguen [33] has studied the issue of component integration for large-scale systems, and proposes a module interconnection language (LIL) with a program methodology for composing software components that facilitates reuse. Goguen introduces three semantic concepts: (1) *theories*, which, like Larch traits and the MTS definition of component specifications, associate semantic descriptions with software components; (2) *views*, which describe implementation alternatives for software interfaces; and (3) *horizontal* composition, which describes structural alternatives for a given level of an abstract machine. The MTS notation was adapted to include a construct for views to enable the software designer to express how the components are related through consistency relations. Traditional transformation techniques deal with *vertical* composition, which refines programs to a lower level of abstract machine. The MTS transformation techniques for integration is an example of horizontal composition that is necessary for structuring large-scale programs. LIL is descriptive, and describes what the alternatives are, but is not constructive, offering no guidance in implementing the alternatives. The techniques developed in this thesis describe how to use transformations to supplement the horizontal composition methods using a constructive approach.

Tracz develops a system called LILEANNA [89] based on LIL using Ada as the programming language extended with ANNA [56], a specification language for Ada that allows the insertion of annotations. Tracz provides a model of module composition, fully defines the composition mechanism (in LIL) by allowing horizontal, vertical, and generic instantiation in module expressions, and discusses the relationship between the various types of module structuring. This provides a framework for describing changes to modules and may give insight into new transformations and the structures needed for automation. The requirements for composing modules in LILEANNA makes the underlying assumption that modules that share data have the same data representations. The MTS methods provide complementary techniques for integrating multiple data representations.

The Draco [62] system aids the software designer in constructing software systems from reusable software parts. The software designer uses a *domain language* for describing programs in each problem area. The objects and operations in a domain language represent analysis information about a problem domain [26]. There is one software component for each object and operation in the domain. Objects and operations from one domain language are implemented by being modeled by the objects and operations of other domain languages. Eventually, the developing program is modeled in a conventional executable (programming) language. Programs are constructed from the objects and operations of a suitable domain. The use of a domain language aids the software developer in expressing the problem in the

language; however, support is lacking in structuring objects and operations into datatypes or objects (in the object-oriented sense). For example, an operation and an object that it manipulates are refined separately (since each one is represented as a separate component); furthermore, the refinements must agree on the underlying representation of the object. The MTS methodology, on the other hand, provides a systematic means to reach agreement of the representation of the object and to derive new implementations of operations defined on the object.

Draco transformations represent optimizations at the domain language level; Draco refinements make implementation decisions. Traditional transformations combine the Draco notions of transformations and refinements. The MTS methodology similarly makes a distinction between using transformations within and between domain levels. The techniques for integration, in effect, stay within the same domain. In contrast, rather than source-to-source transformations which consist of a lefthand side that is replaced by a righthand side, MTS uses transformations to derive and manipulate module interfaces. The subsequent refinement steps make choices to implement the program in a more specialized domain. Refinement in Draco is an interactive process where the systems designer is involved in making decisions. For large systems, there are far too many decisions for the systems designer to make. Draco provides two mechanisms for dealing with this complexity: domains and tactics. The systems designer need only work in one domain at a time which limits the scope of what to think about. Tactics limit the number of decisions that must be made. Even with these mechanisms, as systems are built and adapted, a Draco system may become increasingly difficult to maintain. The software developer using MTS for refinement, on the other hand, uses a small set of well-defined transformations for adjusting interfaces and data representations.

7.3 Defining and Managing Representations

In this section, techniques used for defining and managing representations are evaluated with respect to the criteria for appropriateness of representations, interface agreement, and adaptability. Evaluating the appropriateness of representations raises the question, how do chosen data representations reflect the requirements of each component? As a system evolves, compromises are inevitably made to data representations in order to meet diverse needs; often expedient solutions are developed from which a later retreat is required. Evaluating interface agreement raises the question, when must agreement on interfaces in the design of software be reached? Delaying design decisions when *a priori* agreement cannot be reached may make the design process easier initially but additional work is usually required to integrate the components later. Evaluating adaptability raises the question, how easy is it to incorporate new components into the system?

The *views* approach of Garlan [29] has motivations similar to the MTS approach; rather than having to decide in advance on some compromise representation, separate components with appropriate representations are designed and later integrated. This allows agreement on interfaces to occur later in the design process. Unlike the MTS approach, however, merging is restricted to a small number of fixed datatypes, thus yielding a greater degree of

automation at the expense of expressiveness, power, and flexibility. The *programming with views* approach of the Gandalf group [39] extends Garlan's work to support the integration of programs (tools) that access and manipulate a shared data representation. The description of the shared data representation is factored into the individual tools; each tool defines its own view of the data structure it uses. These views are later integrated by describing how information is shared between tools and what invariants must be maintained between different views of the same data. This supports the merging of arbitrary abstract datatypes (that are connected via "compatibility maps"), but is less automatic since it requires all operations to be rewritten by hand for a merged type.

The MTS techniques for deriving module interfaces require a single compatibility map to interconnect any two components. This facilitates the design level of integration, but requires additional work at the implementation level of integration where the developer must provide some guidance. The integration of the components is resolved at compile-time under MTS but at runtime under Gandalf. A static approach permits optimizations to be performed as well, and alleviates the additional overhead required to handle integration during runtime. It is important to keep in mind that the MTS techniques were designed to handle a slightly different problem (which permits this static approach), building a datatype from a collection of pieces and not maintaining separate views in the merged type. However, these techniques may provide a basis for formalizing the process of merging in programming with views (see *programming with views* in Section 7.6 for more details).

Whenever a change is made to a component, this may affect other components that in some way depend on it. The techniques for manipulating module interfaces isolate change in the new component and then propagate change to other interdependent components through the process of integration. Research in databases has similar motivations for introducing and propagating change. Balzer [2] provides a language for making structural changes to the description of a knowledge representation system. He also provides tools for mapping those changes into corresponding transformations on the existing data. The change is immediately propagated. An alternative to propagating the change immediately is to use versions so that, for example, types of objects stored in a database can be changed without having to modify existing data or tools [81].

The coexistence of old and new data and tools comes at the cost of the additional overhead for maintaining and accessing multiple type versions, change is never propagated. An intermediate approach is to propagate structural changes only when data is actually accessed by a system configured for the new structures. TransformGen [85] was designed to solve the problems of grammar evolution for structure-oriented environments. The implementor makes structured changes to the grammar of a structure-oriented environment. The output from TransformGen is a new grammar together with a transformer, which takes instances of database trees built under the old grammar and automatically converts them to instances of database trees that are legal under the new grammar. In MTS, the changes are propagated immediately, however, since changes are isolated in the new component, the original system (in effect, an earlier version) is still available.

7.4 Data Design and Refinement

In this section, frameworks and programming environments are evaluated with respect to the criteria for correctness, performance, and automation, to determine how well they support the process of data design and refinement. Evaluating correctness raises the question, how can higher assurance of correctness be provided for larger systems, or, rather, for aspects of behavior of larger systems? Evaluating performance raises the question, what choices are available to the designer for improving the efficiency of a program? These might include techniques that affect the frequency of execution of parts of the program and how readily information is made available. Evaluating automation raises the question, what elements of the process of producing efficient programs from high-level programs can be mechanized? The main emphasis here should be on achieving productive interactions of automated systems with developers and maintainers.

There are a number of *formal frameworks* for developing larger-scale programs by transforming high-level specifications into executable code. Like the MTS approach, they seek to extend transformation techniques to larger-scale systems, and similarly involve the software designer in the design of data representations (usually in order to avoid limiting the expressiveness of the specification language). The developers of CIP [5], for example, advocate using algebraic specifications as the starting point for a top-down method of program development. The developers of Extended ML [73] and VDM [7] use formal verification to invent new implementations and prove them correct. The MTS approach differs from these in its support for the integration of separate components. This gives the designer the flexibility to delay agreement on, as well as adapt, the interfaces of a (module) system.

The Programmer's Apprentice [70] provides assistance in the implementation, design, and requirements phases of the programming task. A formal representation for programs and programming concepts is provided by the Plan Calculus. A plan is a generalized representation of a program and is represented as a graph structure consisting of boxes and arrows. The boxes denote operations and tests; the arrows denote control and data flow. The representation has a graphical notation and a formal semantics used for reasoning. Relationships between plans (*eg.*, specification and implementation) is represented by an overlay. An *overlay* defines a mapping from the set of instances of the implementation plan to a set of instances of the specification. It is a generalization of the abstraction function in the abstract datatype methodology. Like overlay's, MTS translation functions represent relationships, but relationships between components within a given level of abstraction, rather than relationships between different levels of abstraction.

The Design Apprentice (a subsystem of the Programmer's Apprentice) is designed where: (1) a task is expressed in a declarative (specification-like) input language; (2) the system provides detection and explanation of errors made by the programmer, (completeness and consistency); and (3) the system automatically selects reasonable implementation choices. Information is embodied in "cliches" (combinations of program elements). Perhaps the methods developed in this thesis could be used to build cliches in the first place. Then the MTS methodology could use cliches as a method for reuse.

The MTS methodology allows the designer to choose the data representation and to customize it in the context that it appears in, unlike other methods where representations and optimizations are chosen from a predefined set. SETL [79] is an example of a very-high-level language based on set-theoretic syntax and semantics. For the most part, the SETL user is not involved in choosing data representations. Instead, they are automatically chosen by a compiler for each abstract object, based on a catalog of optimizations. The optimizer does not make any significant change to the algorithmic form of the program, it being more concerned with the data representation that supports the specified algorithm. Wide-spectrum languages such as Refine [82] give the software developer more control by providing an interactive environment where the developer can apply optimizations based on the usage and context of the data. The optimizations are drawn from a set of supplied rules and templates. OBJ [34] can be considered an executable specification language. It provides a notation for structuring algebraic specifications into hierarchies of parameterized modules [28]. Rather than focusing on data representations, the OBJ system implements an equational rewriting system.

The MTS techniques also have important differences with the transformation-based approach of Hisgen [45], which seeks to optimize user-defined datatypes automatically by using predefined type-specific transformation rules. The programmer writes pre-conditions and post-conditions as well as transformations for each operation. Hisgen's mechanism is well suited to the development of customized types for use in a very-high-level language system. The type-specific optimizations provided by a type designer can be applied automatically by a compiler, but the type designer is responsible for their correctness. Instead of anticipating the collection of transformations needed in advance, the software developer using MTS has a small collection of well-defined transformations to use to customize the datatype in the context that it appears in.

In *program synthesis* by Manna and Waldinger [57] programs are extracted from a constructive-style proof. The proof serves as a high-level language. Recent developments in this approach to manipulating proofs include program development through proof transformation [68]. While this provides a solution to the problem of designing the initial program, there is still the need for a complementary process that optimizes the extracted program by manipulating the data representations and the structure of the interfaces. Transformation techniques could provide this capability [1].

7.5 Adapting Interfaces

In this section, object-oriented techniques and transformation systems that support evolution are evaluated with respect to the criteria for adaptability by means of modifying existing interfaces. Evaluating interface agreement raises the question, when must agreement on interfaces in the design of software be reached? Delaying design decisions when *a priori* agreement cannot be reached may make the design process easier initially but additional work is usually required to integrate the components later. Evaluating adaptability raises the question, how easy is it to modify existing interfaces and incorporate new components into the system?

Traditional *object-oriented* techniques [87] enable the development of customized abstract interfaces based on existing types, but with representations and implementations shared among the variants (*eg.*, through inheritance). That is, in the object-oriented world, a "specialized type" of one or more existing types may have a specialized abstract interface, but the underlying implementation will fundamentally be determined (through inheritance) by the existing type implementations. This can limit the efficiency of the implementation and has led some languages to allow violations in abstraction boundaries by permitting access to the representation of the ancestors of an inherited object [83]. A more controlled approach is the "friends" declaration in C++ [86] where classes have special access to other classes that are declared friends. The research presented in this thesis, however, may provide a means to obtain truly specialized implementations for the specialized types by means of program transformations. The MTS techniques provide optimization and integration of multiple representations to complement the object-oriented approach, which supports flexible integration and enhancement but requires compatible interfaces.

Griswold's work [36] on *program restructuring* as an aid to software maintenance uses a transformational approach to address a different problem. He introduces a set of automatable transformations to manipulate the structure of a system for supporting evolution through manipulation of program structure. These transformations are applied locally to effect a syntactic change; the system may make non-local changes to preserve data flow dependence and control flow dependence. Griswold's work focuses on the transformation of the syntactic constructs of a block-structured language, whereas MTS is focused at the module level.

7.6 Applications for the Module Transformation System

Domains where this approach is useful include: the development and evolution of prototypes; the reuse of software modules in libraries; programming with views; heterogeneous systems; and type management for persistent objects.

Development and Evolution of Prototypes. In the development and evolution of prototypes, a principal objective is achieving functionality in the easiest possible way. For a complex type in a system (*i.e.*, abstract datatype interface) with many operations associated with it, it might be difficult to design a single representation that is suitable for prototyping and exploratory development of all of the various operations of the abstract datatype. In such cases, it might be easier to design a collection of separate representations that work for various subsets of the full set of operations. Individual operations could thus be separately prototyped using appropriate representations. The problem then is to assemble the various operations and conflicting representations into a single type that can be merged into a larger prototype system and possibly transformed into a high performance implementation.

Reusable Libraries of Software Modules. These transformation techniques may also enhance the ability to create and retain software objects for reuse [50] (*eg.*, the Larch [38]

library of abstract interfaces). One factor that makes reuse hard to realize is that multiple developers of system components with shared abstract interfaces will evolve data representations that conflict. The conflicts are not due to a failure to settle interfaces in advance, but due to technical needs that motivates choices by separate implementors (*eg.*, the context in which an operation is used, frequency of use, space/time tradeoffs desired, or flexibility/efficiency tradeoffs desired). Because of this conflict, components cannot be shared unless additional components are implemented that translate representations. The cost of this translation is usually too high except in a prototype.

This research explores reuse through customization [23, 76], and provides a framework for the adaptation of datatypes and the manipulation of interrelated modules. An instance of this framework is a network of generalized and specialized versions of abstract data types where the MTS technique builds and maintains the network.

Programming with Views The Janus system [39] supports the merging of abstract datatypes. The techniques developed in this thesis may help formalize this process of merging where transformations "compile" views into a canonical form and specialize them. Janus provides four kinds of storage models: disjoint, derived, shared, and "anything goes."

In the disjoint storage model the fields of a merged class are the disjoint union of the fields of the component classes. This corresponds in some way to the process of "merging" where the fields of a data aggregate are the cross product of the fields of the components. As with disjoint union, all the fields from each original component appear separately on the merged aggregate. The difference being that separate views are not maintained in the aggregate. Each component contributes to the new single aggregate. It could be possible to implement views on top of this.

In the derived storage model the fields from one of the original classes are not stored explicitly. Instead their values are dynamically calculated when needed. This corresponds to the process of "translating" where the fields from one component are not stored but calculated from other fields in the aggregate. This could be done dynamically via a translation function, or statically by deriving new implementations of the component based on a new representation that is stored in the data aggregate.

In the shared storage model two fields from the original classes are stored as a single field in the merged class. This is a special case of "translating" where the fields from one component are not stored but calculated from the other fields in the aggregate. Since the data is identical, then the operations can be optimized to manipulate the shared data directly.

In the "anything goes" storage model the merged class contains fields that have no direct correspondence with fields from any of the original classes. This corresponds to the specialization step of deriving an efficient implementation from the prototype. It is also possible that a more generalized implementation could be derived [21]. There is in fact a correspondence which is recorded in the derivation in terms of the design decisions and transformation steps taken to produce the new data aggregate.

Heterogeneous Systems. The draft report on a Common Prototyping System [3] identifies requirements for a language and system that supports prototyping as a first step towards

building heterogeneous systems. The report identifies the need for "multi-language interoperability" where fragments of existing code are "composed" in a prototype. One subproblem involves converting data items from one representation used in one language to the representation used in the other. Hayes and Schlichting [44] have developed a solution to this data representation problem in multiparadigm programming for a fixed set of standard datatypes. Perhaps data aggregation can provide a solution for user-defined datatypes as well.

Type Management for Persistent Objects. How does one propagate the effects of changing a type to instances in a persistent store? A key subproblem is updating the library of persistent objects [16]. One solution is to use a view-based model, and the MTS approach could systematically develop the merged type representations. Another solution is to emulate the new behavior, and the MTS approach could aid the process of finding a type representation that could support the functionality of emulation.

Chapter 8

Conclusions

The organization of interfaces among system components is a key task in the construction and management of larger-scale software systems. Datatypes and modules that interact must agree not only on the abstract interfaces, but also on data representations if their implementations share data. As a software system evolves, the need to adapt existing interfaces can arise. Thus, this problem of integration persists for as long as the system is maintained.

This thesis has demonstrated that program transformations (semantics-based program manipulation) provide systematic support for integrating general-purpose software modules into efficient systems. This approach also provides support for adaptive and perfective maintenance. Complex type definitions initially consist of a number of components that are composed via translation functions and module extensions. The initial interfaces are then integrated, resulting in complex composite interfaces, by using data aggregation transformations.

8.1 Contributions

This thesis has three major contributions: (1) New transformation techniques for data aggregation that enable the application of transformation techniques to the development and adaptation of larger-scale systems. (2) A hand-conducted study of the derivation of a display editor that illustrates a proof-of-concept for the methodology. (3) A framework for describing data transformation techniques that enhances the understanding of the terms used informally, provides structure to aid the software designer in using the approach and is an important step towards automating the system.

Larger-Scale Systems. The first contribution, new transformation techniques for data aggregation, enables the application of transformation techniques to the development and adaptation of larger-scale systems. Figure 8.1 is an abstract view of this process for constructing systems using the module interface transformation system, and corresponds to the enumerated steps listed below. The process starts with the *design* of the top-level aggregate specification (step 1). The software designer who wishes to design a complex

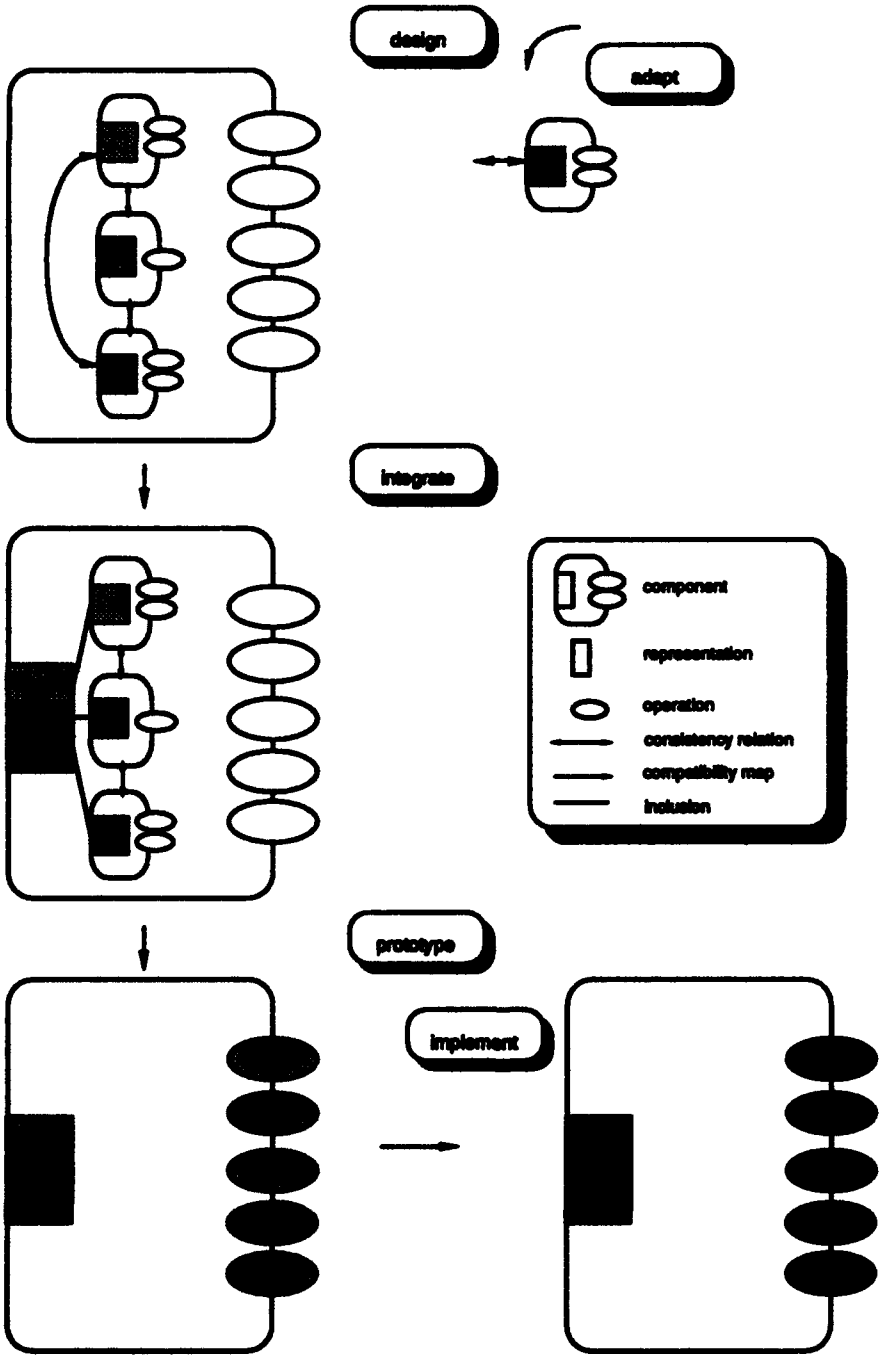


Figure 8.1: Strategy for Constructing Systems

system is able to decompose the problem into components that best model that portion of the problem. The components may be obtained from a library of software assets of standard interfaces or prototyped by the designer using a data representation that most closely models the subproblem. Consistency relations establish correspondences among the data representations. This aggregate specification is used in the *integration* phase to produce an aggregate definition (step 2). Obtaining the aggregate definition from the aggregate specification is a mechanical process once compatibility maps (which respect the consistency relations) are provided. The aggregate definition is in a format upon which data translations can be performed to obtain an executable *prototype* (step 3). Then additional transformations such as *expose*, *incorporate*, *release*, and *shift* can be performed to optimize the prototype into an efficient *implementation* (step 4). Later on the software designer may wish to introduce additional functionality in the *adapt* phase.

The essential steps for implementing datatypes or modules by components:

1. *Structure (compose) the system using modules.* List the operations that constitute the requirements for some system. Define the component implementations, each of which implements some subset of the interface. Collectively, all of the components implement the entire interface. Use module extensions to adapt the abstract interfaces. Establish any data invariances among the components by defining functions that translate from one component representation into another to establish the consistency of the collection of data representations.

2. *Integrate the components to define the datatype or module.* Choose as an "expedient" representation the product of the component representations. Each operation defined in a component induces a corresponding operation on the composite datatype or module. Each operation definition is put into a format amenable to transformation.

3. *Derive the first executable prototype of the datatype or module.* Since the definitions are data transform procedures, this is done by applying transformations. The expression procedures are transformed into functional definitions. When adapting the system, it may be possible to reuse some of the information from the derivation of the integration of the original system.

4. *Derive an efficient implementation by translation.* Uncover an efficient representation by eliminating unnecessary redundancy and specializing data in the context that it appears in. Derive efficient implementations of the operations on the new representation using transformations.

Using these techniques suggests a paradigm for datatype implementation by "components." Sometimes it is difficult to design types or anticipate future needs. Instead of introducing a type and anticipating all necessary operators, the operations are designed as we discover the need for them in the program using the datatype. The representations are selected based on the needed operations. This thesis also suggests a paradigm of system implementation by modules where we use transformation techniques to get better performance than simply reusing code. The module transformation system provides a way to manipulate the modules and to change the cohesiveness and the couplings of the modules.

These paradigms provide benefits to scaling primarily at the design level. Complexity is managed through abstraction, modularization, and step-wise transformation. The focus of the software designer is on the design domain. These design decisions are translated into changes throughout the system at the integration implementation level to integrate and optimize the system. The formal manipulations at this level are generally carried out within local contexts. However, in order to claim that this method truly scales, then assistance is needed at the integration implementation level in carrying out all of the steps. Most of these are mechanical steps that could be performed with automated support.

More experience using the techniques may lead to interesting object-oriented applications. Object-oriented programming supports flexible integration and enhancement but requires compatible interfaces. These techniques provide optimization and integration of multiple representations so the two techniques are complementary. This could be done initially within the notation based on Standard ML, since objects with state can be defined using ML functors, and a simple inheritance mechanism is supported.

Proof-of-Concept. The second contribution, a hand-conducted study of the derivation of a display editor, illustrates a proof-of-concept for the methodology. This thesis shows the hard problems of module interface integration that occur in software development and introduces a methodology that complements or enhances existing methods for solving them. The advantages of using this methodology include the ability to delay decisions, to have the system infer some of the information for the implementation from the design, and to reuse the insights and transformation steps from previous developments.

In order to demonstrate the techniques for integrating module interfaces by program-transformations, a simple interactive display-editor was developed. First a text buffer was implemented and then additional functionality was introduced to demonstrate how the text buffer is adapted. The decision for the data representation of the buffer was delayed until after the integration process. The components were simply connected (via compatibility maps); the methodology provided a systematic means to infer the other connections when they were needed. During adaptation, a single connection between the new component and the existing system was needed; once the new representation was integrated with the connecting component from the existing system, the integration with the rest of the system can make use of existing insights and transformation steps. By going through the exercise of constructing a buffer from components and then adding additional components to adapt the buffer, a lesson was learned that components can implement parts of a datatype and that the transformation methods enable the integration of the parts into an aggregate data structure.

Next the focus of the example changed to the module level and the buffer was used as a part of a larger display-editor system. The lessons learned included how transformation techniques can be applied to hierarchically structured module systems, where modules are defined in terms of other modules. The buffer datatype that has been previously developed is reused and customized in the context of a larger interactive display-editor. Transformations are used for adapting data representations and abstract interfaces, and for optimizations.

While using the module interface transformation system, the focus of the software designer is on the design domain and away from the application of the technique (which is the traditional transformation approach). At the design phase, the system is composed from components with straightforward implementations. During the integration process, the insights deal with reasoning about the domains in which the components are modeled.

Automating the technique is an important step to accomplish next, so that more experience in this and other domains can provide additional information about the applicability of the techniques and the costs associated with using them.

Framework. The third contribution, a framework for describing data transformation techniques, enhances the understanding of the terms used informally, provides structure to aid the software designer in using the approach and is an important step towards automating the system. A notation based on Standard ML [59] modules is used to represent the components of a system. In addition to representing datatype definitions and modules, the notation needs to also express the other structures in the transformation process. Required extensions to the notation to express the transformation process include: axioms [72], views [33], expression procedures [74], and a means to express alternative solutions. Axioms are needed to enrich the expressive power of Standard ML so that the properties of the aggregate specification can be defined. Views enable the software designer to express how the components are related through consistency relations. Expression procedures provide the notation needed to express the initial definition of the aggregate upon which module transformation rules can be applied. Alternative solutions are used to maintain the internal consistency of the aggregate.

As progress is made in explaining the techniques in terms of a framework, it may be possible to treat the derivation process as an object that can be formally manipulated and to capture the insights from the software designer. This increases the potential for reusing the previous derivations in integrating the existing system, when adapting the system by adding a new component. This was done informally in the editor derivation, by reusing data transform procedure definitions, the structure of the transformation steps, and some of the insights provided by the software developer. See Baxter [6] and Cheatham [15] for formal approaches. One way to manipulate or optimize the process would be to apply the techniques to the compatibility maps, for example, compile a lengthy sequence of compatibility maps into a single compatibility map or produce the transitive closure to get connections between all of the components starting with a collection of components that are simply connected. It also may be possible to generate the inverse compatibility maps as well (when they exist). Another optimization is to synthesize specialized compatibility maps between the components that comprise the aggregate from the compatibility maps among the original components and the refinement step (which can be viewed as a translation function from the aggregate prototype to the aggregate implementation). Of course, differing strategies could be devised to obtain these new compatibility maps. Rather than obtaining them all at once, it may be beneficial to generate them only when needed, or on demand.

The translation function need only respect the consistency relation, by translating one component representation into another representation. During the derivation process, the

software developer provides insights to enable the construction of alternative implementations of a component. These insights, in effect, are a database of information about the properties of the data and their interdependencies. Perhaps these interdependencies could be formalized as partial translation functions (with information obtained from insight steps and the properties of operations) that capture the insights provided by the software developer. As the database of information is built up, the translation function may eventually capture all of the interdependencies among the data to fully implement the consistency relation. This provides a mechanism for reusing the insights from the software designer, and in some cases, constructing the inverse translation function, which greatly simplifies the transformation process.

The prototype and implementation stages may not have to be two distinct steps. Often a great deal of work is done in the prototype stage only to be discarded later on. It may be possible to use a form of filter promotion to prune out the derivations that are unlikely to be fruitful, or, to use a form of lazy derivation, or derivation on demand so that the software developer only expands enough of the derivation that is needed.

8.2 Turning the Method into a Software Reality

A formally-based methodology has been devised for systematically integrating software components, through the mediation of abstract interfaces and underlying data representations. This provides for the ability to delay or revise design decisions when it is difficult to reach an *a priori* agreement on interfaces or data representations. A proof-of-concept for this methodology has been demonstrated by the derivation of an interactive display-editor. However, this method is not yet a software reality: (1) Connections need to be established with software process models. The methodology of the module transformation system could help reduce the risk that a software system does not conform with the actual need, by providing earlier validation within a software process model. The software process model could provide more guidance to the software designer constructing software systems using the module transformation system. (2) Automation is an important step in making progress toward turning the concept into an engineering method. (3) A more formal model of the transformation system would aid in the explanation and automation of the techniques. (4) Integration is low-level; notions of consistency at a higher level of abstraction than translation functions may provide better models for explaining the design and manipulation of module interconnections.

8.2.1 Software Process Models

Connections need to be established with software process models. The methodology of the module transformation system could help reduce the risk that a software system does not conform with the actual need, by providing earlier validation within a software process model. The software process model could provide more guidance to the software designer constructing software systems using the module transformation system.

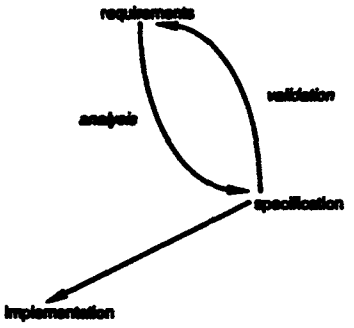


Figure 8.2: An Abstract View of Traditional Software Development

Reducing the Risk of System Nonconformance with the Actual Need. Process models have been developed to organize the stages of software development. The waterfall model is the basis for most software development in government and industry. Development is divided into stages (*eg.*, requirements analysis, specification, coding, testing). There is feedback between successive stages and a form of prototyping through the building of an initial prototype in parallel with requirements analysis. While the waterfall model may work well for problems that have a well defined domain, this traditional approach to developing software (Figure 8.2) – requirements, analysis, specification, validation – does not always yield the desired result because often all the requirements are not known in advance, but require some experimentation. This is true whether the implementation is designed separately and then verified or is derived using formal techniques. This entails high risk because formalized requirements are not validated until very late.

In any software development there are two important conceptual points. The point of making a decision (*eg.*, design decisions, data structure commitments) and the point of learning the consequences (*i.e.*, the validation of the earlier decision). The interval in between is a measure of the risk involved. The longer the interval, the greater the risk that the initial decision may not be the correct one which will affect subsequent decisions that are based on the initial decision. The way to reduce risk is to try to bring the two points closer together, either by delaying decisions (*eg.*, abstracting function) or by advancing validation (*eg.*, building prototypes or reusing validated components). There are constraints on how much the points can be shifted. Delaying decisions cannot be done indefinitely and too much delay may produce over generalized results that are inefficient. Likewise advancing validation is feasible only in the context of given resources.

The spiral model [8] creates a risk-driven approach to the software process. It accommodates the good features of other software models (*eg.*, the waterfall, evolutionary, and transform models) while avoiding many of their difficulties. Software process models can be improved by developing more support for the incremental creation and adaptation of systems (Figure 8.3) using formal methods (cf. inferential programming [77, 51]). Requirements can be used to produce prototypes that provide feedback for early and incremental

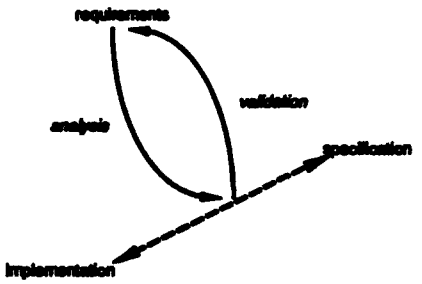


Figure 8.3: Evolutionary Transformation

validation and hence reduce risk. Derivations are bi-directional — a prototype can be refined towards an implementation or "generalized" towards a more formal specification (see work by Dietzen and Scherlis [23] for some thoughts on generalization). Such a method would support an incremental, evolutionary approach to prototyping and systems development that is both adaptable and driven by risk requirements. It would also focus on the *stepwise* attainment of appropriate levels of abstraction, function, and scale for complex interfaces and complex systems.

This thesis research incorporates techniques from inferential programming with a narrower focus on prototyping and a wider focus on scaling and adapting interfaces. The research assembles a set of techniques that supports: (1) creation of prototypes that involve complex typed objects; (2) adaptation of these prototypes, including restructuring of type signatures and addition of new representations for newly delineated operation subsets; and (3) evolution of prototypes into efficient implementations, hence reuse of prototype fragments through aggregation and customization.

Still, there are numerous issues to be addressed for such a model. What structure will derivations take? How are derivations manipulated? Will a new high level language be required? Before some of these broad issues can be addressed, it is useful to have some practical experience with some examples. This thesis contributes by exploring a "vertical slice" of the issues that is narrow and deep, in order to expose some of the research issues and focus on the overall structure. It examines the relationships among requirements, prototypes, and implementations. Therefore, some *ad hoc* decisions have been made regarding derivation design structure and language, to provide a basis for more detailed future research in each of these areas of the model.

Guidance in Constructing Systems Using this Approach. Connections could also be established with software process models to provide more guidance in constructing software systems using the module transformation system. Software process models that support prototyping and evolution, such as the Spiral Model [8], could provide guidance in evaluating the choices made available by the approach developed in this thesis.

The "slice" of the derivation model (Figure 8.3) that is developed in this thesis provides an overall methodology to guide the software designer in the design and development process (Figure 8.1), and criteria such as the cost of integration and implementation are available to evaluate the consequences of making certain decisions (see Chapter 5).

The methodology provides a structure supporting development. It also records the design decisions made during the integration process which are useful for later modification, adaptation, and maintenance. Still, an "application theory" is missing, and must be applied externally by the software designer. The application theory is not part of the methodology because all of the information is not available in the module system itself. Many decisions are based on the environment in which the system is to be used and the non-functional requirements of the user. Such information is available from the software process model. Within such a process model, this methodology provides a specific capability where the software process model provides advice on how to proceed [93].

8.2.2 Automation

Automation is an important step in making progress toward turning the concept into an engineering method. Chapter 5 provided an interpretation of the results of the editor derivation. The benefits these techniques offer occur primarily at the design level. In order to make this method accessible, assistance is needed at the implementation level in carrying out all of the steps. Many of these are mechanical steps that can be performed with automated support.

An initial project would demonstrate the implementation of a program derivation system for a simple functional language (such as a subset of Standard ML) supplemented with expression procedures. This program derivation system could be constructed using the Ergo Support System [52]. Automated assistance is available in the form of tools that store and display the programs and apply the selected transformations. As more information is learned about this process, more of the information provided by the designer may be shifted to the tools, for example, by building strategies out of transformation steps and implementing them as metaprograms. Here are the relevant components for an initial project:

Syntax Facility. Given a BNF-like grammar for the language, the syntax facility [22] produces a parser, lexer, and unparser for translating the program into an abstract syntax tree, manipulating the abstract syntax of the program, and displaying the program on a screen.

Analysis Facility. Given an attribute grammar based on the abstract syntax of the language, the analysis facility [64] produces an analyzer to compute the attribute values of a program. This is useful for doing data flow analysis [63], providing the transformations with additional information about the context (*e.g.*, the *incorporate* and *release* transformations need information about name conflicts and variable references).

Interaction Facility. The interaction facility [27] uses the unparser to display the program on a screen, and allows the user to highlight and manipulate the abstract syntax of the program. This is essential for an interactive approach where the designer needs to point at the program to offer hints. Other views of the process are also provided, such as displaying the derivation as a tree structure.

The derivation is not necessarily a linear progression. Some form of hyper-text system would be useful for navigating through the design choices. MellowCard provides a simple mechanism to navigate through textual information, which is useful for recording the design. There is a need to extend this mechanism to navigate through programs, proofs, and derivations as structures in their own right.

Lambda Prolog. Some form of meta-language is necessary to represent the derivation process, for example, to express the transformations. Lambda prolog [69] is one potential meta-language and has been used to express transformations [40, 41]. The use of higher-order abstract syntax, where the bindings of variables are explicitly represented, is a useful feature for expressing and manipulating the scopes of variables, and for expressing abstraction and application transformations.

Persistent objects. The changes to the program and the recorded derivation (including design decisions) need to be stored for later retrieval. A persistent object database [71] provides storage for these objects that persist beyond the lifetime of any particular application of the derivation program.

8.2.3 Formal Models

A more formal model of the transformation system would aid in the explanation and automation of the module transformation techniques. A framework for the derivation system was presented in Chapter 6. Using a theory of data abstraction and the correctness of modular programming (*eg.*, Schoett [78]) could aid in the explanation and automation of the techniques.

To assert that the module transformation rules preserves the meaning of the datatype, we must have a framework in which to define the meaning of datatypes and operations that can be performed on them. One way to think about the meaning of programs is to use Natural Semantics as used in the definition of Standard ML. For example, $B \vdash P \Rightarrow M$, where, "against the background B, the phrase P evaluates to the meaning M" [61]. When a transformation transforms a program P into a program P', it is possible to check if they evaluate to the same meaning within the same environment. See [47] for a discussion of other frameworks.

Schoett presents a theoretical explanation for the correctness of programs obtained from modular programming using data abstraction. He develops a theory of "cells" which are used to represent the interfaces of modules, and are represented in "design graphs." He uses the model to give a formal definition of decomposition and composition, and of refinement in terms of a correctness relation based on behavioral equivalence.

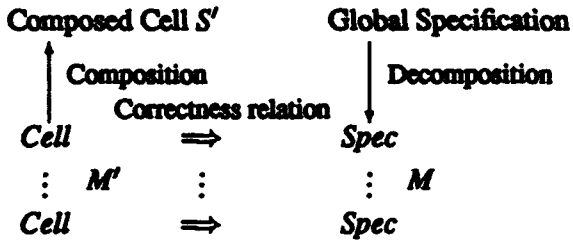


Figure 8.4: Composability of Refinements

A key theorem is the *composability of refinements* (see Figure 8.4). Informally, the theorem states: Let the collection of specifications, M, be a *decomposition* of the global specification, S. Let the collection of implementations, M', be a componentwise *refinement* of M. Then, S', the *composition* of M', is a refinement of S. Informally, the proof of the theorem of the composability of refinements consists of showing: (1) The union of the signatures of M', is a syntactic refinement of the signature of S. (2) Whenever A is a base (*i.e.*, a background or environment) for S then: A is a base for S', there exists a result of S' on A, and every result of S' on A is result of S on A.

Schoett's theory could be extended to allow hierarchical compositions and decompositions in order to show the correctness of transformations on abstract interfaces by moving abstraction boundaries, such as the *incorporate* and *release* transformations defined in this thesis. This could be accomplished by defining what it means to incorporate or release cells in the cell theory. The input and output of the program transformation can then be translated into a family of cells. To show the transformation is correct requires a proof that the family of cells produced by the program transformation satisfies the definition of what it means to incorporate or release cells in the cell theory. Schoett's correctness relation can be used to show the correctness of transformations on data representations, such as the *translate*, *shift* and *expose* transformations defined in this thesis. This is accomplished by demonstrating that the span and unspan functions preserve certain properties to qualify as a proper correctness relation.

Are there a complete set of module transformation rules; are the ones covered sufficient? There are undoubtedly more which will be uncovered as additional experience in using the methodology is applied to other domains. The ones necessary for the integration process have been covered. It would be interesting to look at transforming design graphs and see what module transformations or classes of transformations they suggest.

8.2.4 Integration is Low-Level

Integration is low-level; notions of consistency at a higher level of abstraction than translation functions may provide better models for explaining the design and manipulation of module interconnections. In Chapter 6 the notion of a core component and component specifications were introduced to provide a well-defined meaning to compatibility maps

in terms of consistency relations. It was left unsaid how the component specifications fit together to form the top-level specification. Addressing this issue could lead to an explanation of views in specifications and higher-level abstractions for explaining the design and manipulation of module interconnections.

Translation functions ensure the consistency among components in relation to each other. Their meaning is defined in terms of the consistency relations and the core component. But do we in fact always need to define the core component? We do not need to in the sense that we do not always start with a specification to write a program. But if we want to reason about its meaning, then we need some sort of specification. Once the core component and views of the components are defined, it may be possible in certain cases to derive the translation functions from the views using a syntax-directed translation scheme.

In section 2.2 we discussed the different approaches to looking at correctness diagrams, first looking at verification and then asking whether it is possible to take a constructive approach by using transformations. Behavioral equivalence [78] handles the transition between data specifications and representations in a more general way than abstraction functions. Sannella and Tarlecki [73] develop a methodology for formal development of programs based on this. Perhaps translation functions can be generalized in the direction of behavioral equivalence where transformations are applied to the consistency relations (*eg.*, using real relations as in Prolog). Currently the method is restricted to using functional implementations of these relations (*i.e.*, compatibility maps).

Bibliography

[1] Penny Anderson. Program derivation by proof transformation (thesis proposal). Technical report, Carnegie Mellon University, School of Computer Science, 1990.

[2] Robert Balzer. Automated enhancement of knowledge representations. In *International Joint Conference on Artifical Intelligence*, pages 203–207. ACM, 1985.

[3] Robert Balzer, Frank Belz, Robert Dewar, David Fisher, Richard P. Gabriel, John Guttag, Paul Hudak, and Mitchell Wand. Draft report on requirements for a common prototyping system, November 1988.

[4] Robert Balzer, Thomas E. Cheatham, Jr., and Cordell Green. Software technology in the 1990's: Using a new paradigm. *Computer*, 16(11):39–45, November 1983.

[5] F.L. Bauer, B. Möller, H. Partsch, and P. Pepper. Formal program construction by transformations — computer-aided, intuition-guided programming. *IEEE Transactions on Software Engineering*, 1988.

[6] Ira D. Baxter. *Transformational Maintenance by Reuse of Design Histories*. PhD thesis, University of California, Irvine, November 1990.

[7] D. Bjørner and C.B. Jones. *The Vienna Development Method: the Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[8] Barry W. Boehm. A Spiral Model of software development and enhancement. *Computer*, 21(5):61–72, May 1988.

[9] Grady Booch. Object-oriented development. *IEEE Transactions on Software Engineering*, SE-12(2):211–221, February 1986.

[10] Grady Booch. *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin/Cummings, Menlo Park, CA, 1987.

[11] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[12] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the Association for Computing Machinery*, 24(1):44–67, January 1977.

[13] R.M. Burstall and Joseph A. Goguen. Putting theories together to make specifications. In *Proceedings of Fifth International Joint Conference Artificial Intelligence*, pages 1045–1058, 1977.

[14] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical Report 52, Digital Systems Research Center, Palo Alto, CA 94301, November 1989.

[15] Thomas E. Cheatham, Jr. Reusability through program transformations. *IEEE Transactions on Software Engineering*, SE-10(5):589–594, September 1984.

[16] Steward M. Clamen. Managing type evolution in the presence of persistent instances (thesis proposal). Technical report, Carnegie Mellon University, School of Computer Science, 1991.

[17] John Darlington. The design of efficient data representations, 1980.

[18] John Darlington. The synthesis of implementations for abstract data types. Technical Report 80/4, Imperial College, Department of Computing and Control, 1980.

[19] Peter J. Denning. Beyond formalism. *American Scientist*, 79(1):8–10, January–February 1991.

[20] Frank DeRemer and Hans H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, June 1976.

[21] Scott Dietzen. *A Language for Higher-Order Explanation-Based Learning*. PhD thesis, School of Computer Science, Carnegie Mellon University, Summer 1991.

[22] Scott Dietzen, Mary Ann Pike, and Anne M. Rogers. A guide to the ERGO syntactic processor. Ergo Report ERGO-90-035, Carnegie Mellon University, Pittsburgh, 1990.

[23] Scott Dietzen and William L. Scherlis. Analogy in program development. In J. C. Boudreaux, B. W. Hamill, and R. Jernigan, editors, *The Role of Language in Problem Solving 2*, pages 95–117. North-Holland, 1987. Also available as Ergo Report 86–013, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[24] Conal Elliott. An approach to ADT transformation. Technical report, Carnegie Mellon University, Department of Computer Science, 1986.

[25] Martin S. Feather. A survey and classification of some program transformation approaches and techniques. In L.G.L.T. Meertens, editor, *Proceedings of the IFIP TC2/WG 2.1 Working Conference on Program Specification and Transformation, Bad Toelz, FRG*. North-Holland, November 1986.

[26] Peter Freeman. A conceptual analysis of the Draco approach to constructing software systems. *IEEE Transactions on Software Engineering*, SE-13(7):830–844, July 1987.

[27] Tim Freeman. Overriding methods considered harmful; or ADT-OBJ: Rationale and user's guide. Ergo Report 89–089, Carnegie Mellon University, Pittsburgh, December 1989.

[28] Kokichi Futatsugi, Joseph A. Goguen, Jean-Pierre Jouannaud, and Jose Meseguer. Principles of OBJ2. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 52–66. ACM, January 1985.

[29] David Garlan. *Views for Tools in Integrated Environments*. PhD thesis, Carnegie Mellon University, 1987. Available as Technical Report CMU-CS-87-147.

[30] David Garlan, Gail Kaiser, and David Notkin. On the criteria to be used in composing tools into systems. Technical Report 88-08-09, University of Washington, Dept. of Computer Science and Engineering, 1988.

[31] Susan L. Gerhart. Applications of formal methods: Developing virtuoso software. *IEEE Software*, pages 7–10, September 1990.

[32] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. *An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types*, volume IV, pages 80–149. Prentice-Hall, 1978.

[33] Joseph A. Goguen. Reusing and interconnecting software components. *Computer*, 19(2):16–28, February 1986.

[34] Joseph A. Goguen and Joseph J. Tardo. An introduction to OBJ: A language for writing and testing algebraic specifications. In *Proceedings of Conference on Specifications of Reliable Software*, pages 170–189. Computer Society, 1979.

[35] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Springer-Verlag LNCS 78, 1979.

[36] William G. Griswold and David Notkin. Program restructuring to aid software maintenance. Technical Report 90-08-05, University of Washington, Dept. of Computer Science and Engineering, 1990.

[37] J.V. Guttag and J.J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[38] J.V. Guttag, J.J. Horning, and J.M. Wing. Larch in five easy pieces. Technical Report 5, Digital Systems Research Center, Palo Alto, CA 94301, July 1985.

[39] A.N. Habermann, Charles Krueger, Benjamin Pierce, Barbara Staudt, and John Wenn. Programming with views. Technical Report CMU-CS-87-177, Carnegie Mellon University, Computer Science Department, January 1988.

[40] John Hannan and Dale Miller. Enriching a meta-language with higher-order features. In John Lloyd, editor, *Proceedings of the Workshop on Meta-Programming in Logic Programming*, Bristol, England, June 1988. University of Bristol.

[41] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium, Volume 2*, pages 942–959, Cambridge, Massachusetts, August 1988. MIT Press.

[42] Robert Harper. Introduction to Standard ML. Technical Report ECS-LFCS-86-14, University of Edinburgh, Department of Computer Science, November 1986.

[43] P.G. Harrison and H. Khoshnevisan. The mechanical transformation of data types. Technical report, Imperial College, Department of Computing, November 1987.

[44] Roger Hayes and Richard D. Schlichting. Facilitating mixed language programming in distributed sytems. *IEEE Transactions on Software Engineering*, SE-13(12):1254–1264, December 1987.

[45] Andy Hisgen. *Optimization of User-Defined Abstract Data Types: A Program Transformation Approach.* PhD thesis, Carnegie Mellon University, September 1985. Available as Technical Report CMU-CS-85-166.

[46] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[47] C.A.R. Hoare, I.J. Hayes, H. Jifeng, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Sufrin. Laws of programming. *Communications of the ACM*, 30(2):672–686, August 1987.

[48] Ulrik Jørring and William L. Scherlis. Compilers and staging transformations. In *Thirteenth Symposium on Principles of Programming Languages*, pages 86–96. ACM, January 1986.

[49] Ulrik Jørring and William L. Scherlis. Deriving and using destructive data types. In *IFIP TC2 Working Conference on Program Specification and Transformation.* North-Holland, 1986.

[50] Charles W. Krueger. Models of reuse in software engineering. Technical Report CMU-CS-89-188, Carnegie Mellon University, School of Computer Science, December 1989. To appear in Computing Surveys.

[51] Peter Lee, Frank Pfenning, John Reynolds, Gene Rollins, and Dana Scott. Research on semantically based program-design environments: The Ergo Project in 1988. Technical Report CMU-CS-88-118, Carnegie Mellon University, Pittsburgh, March 1988.

[52] Peter Lee, Frank Pfenning, Gene Rollins, and William Scherlis. The Ergo Support System: An integrated set of tools for prototyping integrated environments. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 25–34. ACM

Press, November 1988. Also available as Ergo Report 88–054, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[53] B. Lientz and E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, Reading, MA, 1980.

[54] Barbara Liskov. Abstraction mechanisms in Clu. *Communications of the ACM*, 20(8):564–576, August 1977.

[55] Barbara Liskov. Data abstraction and hierarchy. *SIGPLAN Notices*, 23(5):17–34, May 1988.

[56] David Luckham and Friedrich W von Henke. An overview of Anna, a specification language for Ada. *IEEE Software*, 2(2):24–33, March 1985.

[57] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):92–121, 1980.

[58] Bertrand Meyer. Reusability: The case for object-oriented design. *IEEE Software*, 4(2):50–64, March 1987.

[59] Robin Milner. The Standard ML core language. *Polymorphism*, II(2), October 1985. Also Technical Report ECS-LFCS-86-2, University of Edinburgh, Edinburgh, Scotland, March 1986.

[60] Robin Milner and Mads Tofte. *Commentary on Standard ML.* MIT Press, Cambridge, Massachusetts, 1991.

[61] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Massachusetts, 1990.

[62] James M. Neighbors. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, SE-10(5):564–574, September 1984.

[63] Robert L. Nord. A framework for program flow analysis. Ergo Report 87–038, Carnegie Mellon University, Pittsburgh, November 1987.

[64] Robert L. Nord and Frank Pfenning. The Ergo attribute system. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 110–120. ACM Press, November 1988. Also available as Ergo Report 88–053, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[65] David Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 14(1):221–227, January 1972.

[66] David Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

[67] H. Partsch and R. Steinbruggen. Program transformation systems. *ACM Computing Surveys*, 15(3):85–94, February 1983.

[68] Frank Pfenning. Program development through proof transformation. In Wilfried Sieg, editor, *Logic and Computation*, Contemporary Mathematics. AMS, Providence, Rhode Island, 1988. Available as Ergo Report 88–047, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[69] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the SIGPLAN '88 Symposium on Language Design and Implementation, Atlanta, Georgia*, pages 199–208. ACM Press, June 1988. Available as Ergo Report 88–036, School of Computer Science, Carnegie Mellon University, Pittsburgh.

[70] Charles Rich and Richard C. Waters. The Programmer's Apprentice: A research overview. *Computer*, 21(11):10–25, November 1988.

[71] Gene Rollins. A platform for expermenting with persistent objects. Ergo Internal Report ERGO-88-074, Carnegie Mellon University, January 1989.

[72] Donald Sannella and Andrzej Tarlecki. Program specification and development in Standard ML. In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 67–77. ACM, January 1985.

[73] Donald Sannella and Andrzej Tarlecki. Toward formal development of ML programs: Foundations and methodology. Technical Report ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, The King's Buildings, Edinburgh EH9 3JZ, February 1989.

[74] William L. Scherlis. *Expression Procedures and Program Derivation*. PhD thesis, Stanford University, August 1980. Available as Technical Report Stan-CS-80-818.

[75] William L. Scherlis. Program improvement by internal specialization. In *Eighth Symposium on Principles of Programming Languages*, pages 41–49. ACM, ACM, January 1981.

[76] William L. Scherlis. Abstract data types, specialization and program reuse. In *International Workshop on Advanced Programming Environments*. Springer-Verlag LNCS 244, 1986.

[77] William L. Scherlis and Dana S. Scott. First steps towards inferential programming. In R.E.A. Mason, editor, *Information Processing*, pages 199–212. Elsevier Science Publishers, 1983.

[78] Oliver Schoett. Data abstraction and the correctness of modular programming. Technical Report CST-42-87, University of Edinburgh, 1987.

[79] E.J. Schonberg, J. Schwartz, and M. Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Transactions on Programming Languages and Systems*, 3(2):126–143, February 1981.

[80] Mary Shaw. Larger scale systems require higher-level abstractions. In *Fifth International Workshop on Software Specification and Design*, pages 143–146. IEEE Computer Society Press, 1989. ACM SIGSOFT Software Engineering Notes, 14(3).

[81] Andrea H. Skarra and Stanely B. Zdonik. The management of changing types in an object-oriented database. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 483–495. ACM, September 1986. SIGPLAN Notices 21(11).

[82] Douglas R. Smith, Gordon B. Kotik, and Stephen J. Westfold. Research on knowledge-based software environments at Kestrel Institute. *IEEE Transactions on Software Engineering*, SE-11(11):1278–1295, November 1985.

[83] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 38–45, September 1986. SIGPLAN Notices 21(11).

[84] J.M. Spivey. *Understanding Z : A Specification Language and its Formal Semantics*. Cambridge University Press, 1988.

[85] Barbara Staudt, Charles Kruegger, and David Garlan. TransformGen: Automating the maintenance of structured-oriented environments. Technical Report CMU-CS-88-186, Carnegie Mellon University, Department of Computer Science, November 1988.

[86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[87] Bjarne Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3):10–20, May 1988.

[88] Bernard Sufrin. Formal specification of a display editor. Technical Report Technical Monograph PRG-21, Oxford University Computing Laboratory, Programming Research Group, June 1981.

[89] William Joseph Tracz. *Formal Specification of Parameterized Programs in LILEANNA*. PhD thesis, Stanford University, 1991. In Preparation.

[90] Anthony I. Wasserman, Peter A. Pircher, and Robert J. Muller. The object-oriented structured design notation for software design representation. *Computer*, pages 50–63, March 1990.

[91] David S. Wile. Type transformations. *IEEE Transactions on Software Engineering*, SE-7(1):32–39, January 1981.

[92] Jeannette M. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):8–23, September 1990.

[93] Jeannette M. Wing and Amy Moorman Zaremski. *Unintrusive Ways to Integrate Formal Specifications in Practice*, volume 551 of *Lecture Notes in Computer Science*, pages 545–569. Springer-Verlag, New York, 1991.

[94] Niklaus Wirth. *Programming in Modula-2*. Springer-Verlag, Berlin, third edition, 1985.

Appendix A

Notation

Program notation. Examples use `typewriter` font for data types, lower-case greek letters for type variables, sans-serif font for functions, and *italics* for variables. The product type constructor $\times$ binds more tightly than the function type constructor $\rightarrow$.

Notation	*Description*
$\Leftarrow$	Function definition.
$\langle x, y \rangle$	Tuple constructor.
$s \times t$	Product type constructor.
t^*	List constructor.
$P(t)$	Set type constructor.
$\Longrightarrow$	Logical implication.
$\lambda i : T \,.\, e$	Lambda abstraction.
$f \circ g$	Function composition.
$\backslash$	Domain restriction operator.
$\oplus$	Functional overriding operator.
Abs, Rep	Create an abstraction, reveal the underlying representation.
`Component.`x	Qualified name of a type or operation in a component.
$\{ c_1 \mid \ldots \mid c_n \}$	Alternative solutions.
map_i^j	Consistency relation between components.
$\mathsf{map}_{i \rightarrow j}$	Compatibility map from one component to another.
span, unspan	Translation functions from one component to an aggregate.
v **sat** *equation*	The variable satisfies the equation.

Domain operations. Common domain operations and constants that are used in the definitions of the component operations and the compatibility maps.

Operation	*Description*
[*c*]	Create a sequence.
@	Append.
cons(*x*, *s*)	Add element *x* to the beginning of sequence *s*.
front(*s*), last(*s*)	All but last element and last element of a sequence.
hd(*s*), tl(*s*)	First element and rest of a sequence.
null()	Predicate, is the sequence empty?
rev(*s*)	Reverse a sequence.
'nl'	Newline character.
'L'	Control L character.
'sp'	Space character.
#*s*	The cardinality of the sequence *s*.
s[..*i*]	Subsequence of *s* from the beginning to *i*.
s[*i*..]	Subsequence of *s* from *i* to the end.
s[*i*..*j*]	Subsequence of *s* from *i* to *j*.
s[*i*]	The i^{th} element of the sequence *s*.
[| |]	Set constructor.
$x \in S$	Set membership.
[|*x* | *P*[*x*]|]	The set whose elements satisfy the predicate *P*.
(*x* | *P*[*x*])	An element that satisfies the predicate *P*.

Auxiliary functions. Special purpose functions that are used in the definitions of the component operations and the compatibility maps.

Function name	*Description*
chars2pages(*s*)	Parse a sequence of characters into a sequence of pages.
lines-to-chars(*l*)	Parse a sequences of lines into a sequence of characters.
nlp(*c*)	Predicate, is the character a newline?
nlpos(*s*)	The newline positions in the sequence of characters *s*.
numnl(*s*)	The number of newlines in the sequence of characters *s*.
npages(*s*)	The number of pages in the sequence of characters *s*.
parse(*s*)	Parse a sequence of characters into a sequence of s-expressions.

Appendix B

Glossary

Abstract interface. The exported types and signatures of the operators in a module.

Aggregate definition. An aggregate definition is a refinement of the aggregate specification. The data representation is defined as the product of the component data representations and the operations are defined in terms of the component operations as data transform procedures.

Aggregate implementation. An aggregate implementation is a refinement of the prototype providing an "efficient" implementation of the datatype.

Aggregate prototype. An aggregate prototype is a refinement of the aggregate definition where the data transform procedures have been transformed into functional definitions to produce the first executable system.

Aggregate specification. An aggregate specification is a specification of a datatype constructed from a collection of components and consistency relations. Each operation in a component induces a corresponding operation in the aggregate that maintains the consistency of the components.

Compatibility map. A compatibility map is a function that respects the consistency relation. It translates one component representation into another representation.

Component. A component defines a collection of operations that may or may not constitute a datatype.

Consistency relation. Given a collection of implementations for a common definition, a consistency relation provides a correspondence between the data objects manipulated in one implementation from those in another.

Data transform procedure. Data transform procedures define alternative implementations on data representations. They may take one of two forms:

1. Given a program f using a data representation D and a function, span, that translates elements of the data representation D to elements of the data representation D', we define f' as:

$$f'(\text{span}(d)) \Leftarrow \text{span}(f(d))$$

2. If, instead, there is a function, unspan, that translates elements of the data representation D' to elements of the data representation D, we define f' as:

$$\text{unspan}(f'(d)) \Leftarrow f(\text{unspan}(d))$$

Projection. Projection functions map an aggregate data object to a component object.

Span and Unspan. A span function is a mapping from one component into the aggregate of all reachable components (*i.e.*, connected by compatibility maps). An unspan function is a mapping from some aggregate of components into the component that can be reached by all of them.

Specialization. The specialization process optimizes a program to take advantage of the context that it appears in. This is accomplished through transformations such as *expose*, *incorporate*, *release*, *shift*, and *translate*.

Translation function. A translation function is a function that translates one data representation into another representation. The spanning functions, span, and its inverse, unspan, and compatibility maps are all examples of translation functions.

View. A view defines how an implementation satisfies a specification and consists of a mapping from the sorts of a specification P to the sorts of an implementation I, and a mapping from the operations of P to the operations of I.

Appendix C

Translating Representations

This section enumerates the ellided steps in Section 3.1.5 of deriving an implementation for move-right on the aggregate data structure based on the component implementation of move-right in $\texttt{Buf}_1$. Recall the data structure for the buffer definition.

```
type buf = Buf of
              (int×ch*          — point of editing and text in the buffer
            × ch* × ch*         — characters to the left and right of point
            × (int× int)        — the line and character position of point
            × line*)            — lines in the buffer
```

We start with the definition of move-right from Figure 3.4.

$$\text{unspan}(\text{move-right}(\texttt{Buf}(p,t,l,r,\langle lp,cp\rangle,ts))) \Leftarrow \texttt{Buf}_1.\text{move-right}(\text{unspan}(\texttt{Buf}(p,\ t,\ l,\ r,\ \langle lp,cp\rangle,\ ts)))$$

Recall that move-right was defined in the $\texttt{Buf}_1$ component. The translation function translates the data aggregate into this component representation.

$$\text{unspan}(\texttt{Buf}(p,t,l,r,\langle lp,cp\rangle,ts)) \Leftarrow \texttt{Buf}_1.\texttt{Buf}(\{\ p\ \mid\ \#l\ \mid\ \#(\text{l2c}(ts[..lp-1]))+1+cp\ \},\ \{\ t\ \mid\ l\ @\ r\ \mid\ \text{l2c}(ts)\ \})$$

The compatibility maps have already been unfolded in unspan to simplify the presentation. (l2c is an abbreviation for lines-to-chars.) Since the definition of unspan does not change, it is not repeated below. There are three ways to accomplish the translation of the aggregate into the $\texttt{Buf}_1$ component. Extract $\texttt{Buf}_1$ from the aggregate directly, or use one of the two compatibility maps to translate $\texttt{Buf}_2$ or $\texttt{Buf}_3$ (extracted from the aggregate) into $\texttt{Buf}_1$. All of the three choices must be included here to ensure consistency in the aggregate.

The aim of the transformations is to manipulate the operation to obtain a definition of move-right that operates on the aggregate directly. Though the implementation of move-right (and the other operations) will change, the meaning of the Buf type remains the same. First the unspan operation is mechanically "unfolded" in the body of move-right.

$$\text{unspan}(\text{move-right}(\texttt{Buf}(p,t,l,r,\langle lp,cp\rangle,ts))) \Leftarrow \texttt{Buf}_1.\text{move-right}(\texttt{Buf}_1.\texttt{Buf}(\{\ p\ \mid\ \#l\ \mid\ \#(\text{l2c}(ts[..lp-1]))+1+cp\ \},\ \{\ t\ \mid\ l\ @\ r\ \mid\ \text{l2c}(ts)\ \}))$$

Next, the component definition of move-right is mechanically "unfolded."

unspan(move-right(Buf($p,t,l,r,\langle lp,cp\rangle,ts$))) $\Leftarrow$
  $\mathtt{Buf}_1$.Buf({ $p+1$ | $\#l+1$ | (#(l2c(ts[..$lp-1$]))$+1+cp)+1$ }, { t | l @ r | l2c(ts) }))

We anticipate having to satisfy the constraint that the character index remain within the current line by introducing case analysis for when the point of editing is at a line boundary and the operation crosses the boundary. The point of editing is at the end of a line when the character position is equal to the length of the current line, $cp = \#(ts[lp])$. The true branch of the conditional is specialized to set the value of cp to $\#(ts[lp])$ and the resulting expression is simplified.

unspan(move-right(Buf($p,t,l,r,\langle lp,cp\rangle,ts$))) $\Leftarrow$
  let $lp', cp' =$ **if** $cp = \#(ts[lp])$ **then** $lp, \#(ts[lp])+1$ **else** $lp, cp+1$ **in**
    $\mathtt{Buf}_1$.Buf({ $p+1$ | $\#l+1$ | #(l2c(ts[..$lp'-1$]))$+1+cp'$ }, { t | l @ r | l2c(ts) })

Using unspan, we know how to map the aggregate to the $\mathtt{Buf}_1$ component. If we could obtain the inverse, then deriving a new implementation for operations on the aggregate would be easier. We simply map the aggregate into the component, perform the component operation, and then map the component back into the aggregate. Obtaining the inverse may not be practical since the translation function may not always be one-to-one, and, even if it were, there may be no easy way to derive it. Rather than coming up with the inverse explicitly, it is sometimes possible to use syntactic manipulations and simplifications to in effect, "invert" the translation function. This is accomplished by simplifying the expressions to match the new representation expressed in the translation function.

Take for example, the inversion of l. The translation function and the preceding definition of move-right are shown below where expressions not dependent on l are ellided. The approach is to use simplification rules to manipulate the instances of l in move-right to match the corresponding instances in the translation function.

unspan(Buf($p,t,l,r,\langle lp,cp\rangle,ts$)) $\Leftarrow$
  $\mathtt{Buf}_1$.Buf({ ... | $\#l$ | ... }, { ... | l @ r | ... })
unspan(move-right(Buf($p,t,l,r,\langle lp,cp\rangle,ts$))) $\Leftarrow$
  let ... **in**
    $\mathtt{Buf}_1$.Buf({ ... | $\#l+1$ | ... }, { ... | l @ r | ... })

Using the simplification rule, $\#l + 1 == \#(l$ @ [hd(r)]), we get $\#l + 1$ in the definition of move-right to match $\#l$ in the definition of unspan. This adds a new constraint that after moving right, the new sequence to the left of the point will be the old sequence with the first element of the right sequence appended. Using the simplification rule, $r ==$ [hd(r)] @ tl(r), we get the other instance of l to meet this new constraint. This is where insights about the domain from the developer are needed (in this and the following step).

unspan(move-right(Buf($p,t,l,r,\langle lp,cp\rangle,ts$))) $\Leftarrow$
  let $lp', cp' =$ **if** $cp = \#(ts[lp])$ **then** $lp, \#(ts[lp])+1$ **else** $lp, cp+1$ **in**
    $\mathtt{Buf}_1$.Buf({ $p+1$ | $\#(l$ @ [hd(r)]) | #(l2c(ts[..$lp'-1$]))$+1+cp'$ },
      { t | l @ [hd(r)] @ tl(r) | l2c(ts) })

The motivation for the next step is to satisfy the restriction (from the constraint on the component) that the character position remains within the current line. We use a version of the simplification rule:

$$\#(\mathsf{l2c}(ts\,[..lp-1])) + 1 + \#(ts\,[lp]) + 1 == \#(\mathsf{l2c}(ts\,[..lp])) + 1$$

This rule states that the length of the preceding lines and the current line is equivalent to the length of all lines preceding and including the current line. Instead of moving the character position past the line boundary, we increment the line index and reset the character position to the beginning of the next line.

$$\begin{array}{l}\mathsf{unspan}(\underline{\mathsf{move\text{-}right}}(\mathtt{Buf}(p,t,l,r,\langle lp,cp\rangle,ts))) \Leftarrow \\ \quad \textbf{let } lp',cp' = \textbf{if } cp = \#(ts\,[lp]) \textbf{ then } lp+1,0 \textbf{ else } lp,cp+1 \textbf{ in} \\ \quad\quad \mathtt{Buf}_1.\mathtt{Buf}(\{\, p+1 \mid \#(l \mathbin{@} [\mathsf{hd}(r)]) \mid \#(\mathsf{l2c}(ts\,[..lp'-1]))+1+cp' \,\}, \\ \quad\quad\quad\quad \{\, t \mid l \mathbin{@} [\mathsf{hd}(r)] \mathbin{@} \mathsf{tl}(r) \mid \mathsf{l2c}(ts) \,\})\end{array}$$

Now that all instances of the old representation appear in the context of the new representation, unspan is mechanically "folded."

$$\begin{array}{l}\mathsf{unspan}(\underline{\mathsf{move\text{-}right}}(\mathtt{Buf}(p,t,l,r,\langle lp,cp\rangle,ts))) \Leftarrow \\ \quad \textbf{let } lp',cp' = \textbf{if } cp = \#(ts\,[lp]) \textbf{ then } lp+1,0 \textbf{ else } lp,cp+1 \textbf{ in} \\ \quad\quad \mathsf{unspan}(\mathtt{Buf}(p+1,\ t,\ l \mathbin{@} [\mathsf{hd}(r)],\ \mathsf{tl}(r),\ \langle lp',cp'\rangle,\ ts))\end{array}$$

Choose the solution.

$$\begin{array}{l}\mathsf{move\text{-}right}(\mathtt{Buf}(p,t,l,r,\langle lp,cp\rangle,ts)) \Leftarrow \\ \quad \textbf{let } lp',cp' = \textbf{if } cp = \#(ts\,[lp]) \textbf{ then } lp+1,0 \textbf{ else } lp,cp+1 \textbf{ in} \\ \quad\quad \mathtt{Buf}(p+1,\ t,\ l \mathbin{@} [\mathsf{hd}(r)],\ \mathsf{tl}(r),\ \langle lp',cp'\rangle,\ ts)\end{array}$$

A new implementation of move-right has been derived that operates on the aggregate data structure directly.

Appendix D

Shifting Computation

This section enumerates the ellided steps in Section 3.1.6 involved in shifting computation of newline information in next-line (in the prototype) to the other operations that generate the datatype, such as makebuf and move-right. In the new implementation, next-line is able to look up the newline information it needs directly, while makebuf and move-right need to do extra work to maintain this information.

Recall the data structure for the buffer prototype.

```
type buf = Buf of
                (int×ch*          — point of editing and text in the buffer
              × ch* × ch*         — characters to the left and right of point
              × (int× int)        — the line and character position of point
              × line*)            — lines in the buffer
```

We start with the definitions for the buffer operations from Figure 3.5. We have chosen a solution for show-char and have defined span according to the observations made in Section 3.1.6. The let statement in move-right has been unfolded in order to simplify the presentation.

```
makebuf  ⇐  Buf(0, [], [], [], ⟨0,0⟩, [])

move-right(Buf(p,t,l,r,⟨lp,cp⟩,ts))  ⇐
    Buf(p + 1, t, l @ [hd(r)], tl(r), if cp = #(ts[lp]) then lp + 1 else lp,
                                      if cp = #(ts[lp]) then 0 else cp + 1, ts)

show-char(Buf(p,t,l,r,⟨lp,cp⟩,ts))  ⇐
    t[p − 1]

next-line(Buf(p,t,l,r,⟨lp,cp⟩,ts))  ⇐
    let d = (nlpos(ts))[lp] − (nlpos(ts))[lp − 1] in
        Buf(p + d, t, l @ r[..d], r[d + 1..], ⟨lp + 1, cp⟩, ts)
```

The translation function records the software developer's requirements to optimize the prototype.

```
span(b as Buf(p,t,l,r,⟨lp,cp⟩,ts))  ⇐  Buf'(p, t, i as lp, nl as nlpos(ts))
```

Since the definition of span does not change, it is not repeated in the definitions below. New definitions for the operations are defined as data transform procedures.

```
makebuf' ⇐ span(makebuf)
move-right'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    span(move-right(Buf(p, t, l, r, ⟨lp,cp⟩, ts)))
show-char'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    show-char(Buf(p, t, l, r, ⟨lp,cp⟩, ts))
next-line'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    span(next-line(Buf(p, t, l, r, ⟨lp,cp⟩, ts)))
```

The buffer operation definitions are mechanically "unfolded."

```
makebuf' ⇐ span(Buf(0, [ ], [ ], [ ], ⟨0,0⟩, [ ]))
move-right'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    span(Buf(p + 1, t, l @ [hd(r)], tl(r), if cp = #(ts[lp]) then lp + 1 else lp,
                                           if cp = #(ts[lp]) then 0 else cp + 1, ts))
show-char'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    t[p − 1]
next-line'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    span(let d = (nlpos(ts))[lp] − (nlpos(ts))[lp − 1] in
            Buf(p + d, t, l @ r[..d], r[d + 1..], ⟨lp + 1, cp⟩, ts))
```

Next, span is mechanically "unfolded" on the righthand side.

```
makebuf' ⇐ Buf'(0, [ ], 0, nlpos([ ]))
move-right'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    Buf'(p + 1, t, if cp = #(ts[lp]) then lp + 1 else lp, nlpos(ts))
show-char'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    t[p − 1]
next-line'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    Buf'(p + ((nlpos(ts))[lp] − (nlpos(ts))[lp − 1]), t, lp + 1, nlpos(ts))
```

Since the character position within the current line, *cp*, will no longer be computed, all references to it must be transformed into an expression that uses data that will still be computed (*i.e.*, *p*, *t*, *lp*, and nlpos(*ts*)). The expression $cp = \#(ts[lp])$ is transformed into nlp($t[p]$) which states that checking to see if the character position is at the end of a line is equivalent to checking if the current character is a newline. (The details are omitted for the sake of brevity.) As in the previous derivation, the guiding motivation is to manipulate the definition so that all instances of the old representation appear in the context of the new representation.

```
makebuf' ⇐ Buf'(0, [ ], 0, nlpos([ ]))
move-right'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    Buf'(p + 1, t, if nlp(t[p]) then lp + 1 else lp, nlpos(ts))
show-char'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    t[p − 1]
next-line'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts))) ⇐
    Buf'(p + ((nlpos(ts))[lp] − (nlpos(ts))[lp − 1]), t, lp + 1, nlpos(ts))
```

Introduce a let abstraction.

```
makebuf'  ⇐  Buf'(0, [ ], 0, nlpos([ ]))
move-right'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
    let p,t,i,nl = p, t, lp, nlpos(ts) in
        Buf'(p + 1, t, if nlp(t[p]) then i + 1 else i, nl)
show-char'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
    let p,t,i,nl = p, t, lp, nlpos(ts) in
        t[p − 1]
next-line'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
    let p,t,i,nl = p, t, lp, nlpos(ts) in
        Buf'(p + (nl[i] − nl[i − 1]), t, i + 1, nl)
```

Introduce an abstraction boundary, $x == \text{Rep}(\text{Abs}(x))$. Here, Buf' is used for introducing the abstraction boundary and Rep' for uncovering the representation.

```
makebuf'  ⇐  Buf'(0, [ ], 0, nlpos([ ]))
move-right'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
    let p,t,i,nl = Rep'(Buf'(p, t, lp, nlpos(ts))) in
        Buf'(p + 1, t, if nlp(t[p]) then i + 1 else i, nl)
show-char'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
    let p,t,i,nl = Rep'(Buf'(p, t, lp, nlpos(ts))) in
        t[p − 1]
next-line'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
    let p,t,i,nl = Rep'(Buf'(p, t, lp, nlpos(ts))) in
        Buf'(p + (nl[i] − nl[i − 1]), t, i + 1, nl)
```

Mechanically "fold" span on the righthand side.

```
makebuf'  ⇐  Buf'(0, [ ], 0, nlpos([ ]))
move-right'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
    let p,t,i,nl = Rep'(span(b)) in
        Buf'(p + 1, t, if nlp(t[p]) then i + 1 else i, nl)
show-char'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
    let p,t,i,nl = Rep'(span(b)) in
        t[p − 1]
next-line'(span(Buf(p,t,l,r,⟨lp,cp⟩,ts)))  ⇐
    let p,t,i,nl = Rep'(span(b)) in
        Buf'(p + (nl[i] − nl[i − 1]), t, i + 1, nl)
```

Since all instances of the old representation appear in the context of the new representation, span(b) is renamed to b'.

```
makebuf'  ⇐  Buf'(0, [ ], 0, nlpos([ ]))
move-right'(b')  ⇐
    let p,t,i,nl = Rep'(b') in
        Buf'(p + 1, t, if nlp(t[p]) then i + 1 else i, nl)
```

```
show-char'(b') ⇐
    let p,t,i,nl = Rep'(b') in
        t[p − 1]
next-line'(b') ⇐
    let p,t,i,nl = Rep'(b') in
        Buf'(p + (nl[i] − nl[i − 1]), t, i + 1, nl)
```

As an alternative, patterns are used on the lefthand side to destructure the datatype instead of using Rep explicitly on the righthand side.

```
makebuf' ⇐ Buf'(0, [], 0, [])
move-right'(Buf'(p,t,i,nl)) ⇐ Buf'(p + 1, t, if nlp(t[p]) then i + 1 else i, nl)
show-char'(Buf'(p,t,i,nl)) ⇐ t[p − 1]
next-line'(Buf'(p,t,i,nl)) ⇐ Buf'(p + nl[i] − nl[i − 1], t, i + 1, nl)
```

In the new implementation, the computation is shifted away from next-line so that next-line simply looks up the position of the newlines surrounding the current line directly. The transformation allows one to get from the prototype representation to this one in a controlled manner.

Appendix E

Integrating Components—Proofs

This section contains the proofs for Section 6.1.4. Each section starts with an aggregate definition and shows the details about how the definition satisfies the axioms comprising the aggregate specification.

$$\textbf{axiom}\ \mathrm{proj}_{I_2}(\mathrm{op}(agg)) = \mathrm{op}_2(\mathrm{proj}_{I_2}(agg))$$

$$\begin{array}{lllllll}
\textbf{axiom} & \mathrm{proj}_{I_2}(agg) & \mathrm{map}_2^1 & \mathrm{proj}_1(agg) & \Longrightarrow & \mathrm{proj}_{I_2}(\mathrm{op}(agg)) & \mathrm{map}_2^1\ \mathrm{proj}_1(\mathrm{op}(agg)) \\
\textbf{axiom} & \mathrm{proj}_{I_3}(agg) & \mathrm{map}_3^2 & \mathrm{proj}_2(agg) & \Longrightarrow & \mathrm{proj}_{I_3}(\mathrm{op}(agg)) & \mathrm{map}_3^2\ \mathrm{proj}_2(\mathrm{op}(agg)) \\
\textbf{axiom} & \mathrm{proj}_{I_3}(agg) & \mathrm{map}_3^1 & \mathrm{proj}_1(agg) & \Longrightarrow & \mathrm{proj}_{I_3}(\mathrm{op}(agg)) & \mathrm{map}_3^1\ \mathrm{proj}_1(\mathrm{op}(agg))
\end{array}$$

Product of the Representations

Here is a simple definition for the aggregate where the data structure is the product of the component representations.

Given: op_2, $\mathrm{map}_{1\to 2}$, $\mathrm{map}_{2\to 1}$, $\mathrm{map}_{1\to 3}$, $\mathrm{map}_{3\to 1}$, $\mathrm{map}_{2\to 3}$, $\mathrm{map}_{3\to 2}$

local

$\mathrm{proj}_{I_2}(\mathrm{Agg}(c_1, c_2, c_3)) \Leftarrow c_2$

$\mathrm{proj}_{I_2}^{-1}(x) \Leftarrow \mathrm{Agg}(\mathrm{map}_{2\to 1}(x),\ x,\ \mathrm{map}_{2\to 3}(x))$

in

$\mathrm{op}(agg) \Leftarrow \mathrm{proj}_{I_2}^{-1}(\mathrm{op}_2(\mathrm{proj}_{I_2}(agg)))$

end

We need to ensure that the above definition satisfies the axioms that define the aggregate.

- *Axiom 1.* The first axiom holds because we derived the new definition from it. However, we must justify the assumptions made during the transformation steps about the inverse projection being injective and a left inverse of the projection. In so doing, a definition for the inverse projection is obtained.

  To show that $\mathrm{proj}_{I_2}^{-1}(\mathrm{proj}_{I_2}(x)) = x$, the proof goes as follows:

  $$\mathrm{proj}_{I_2}^{-1}(\mathrm{proj}_{I_2}(\mathrm{Agg}(c_1, c_2, c_3))) \stackrel{?}{=} \mathrm{Agg}(c_1, c_2, c_3)$$

  First unfold proj_{I_2},

$$\mathsf{proj}_2^{-1}(c_2) \stackrel{?}{=} \mathtt{Agg}(c_1, c_2, c_3)$$

and then unfold proj_2^{-1}.

$$\mathtt{Agg}(\mathsf{map}_{2\to 1}(c_2), c_2, \mathsf{map}_{2\to 3}(c_2)) \stackrel{?}{=} \mathtt{Agg}(c_1, c_2, c_3)$$

The components are consistent by definition, so that c_1 and c_3 can be expressed in terms of c_2, that is, $c_1 = \mathsf{map}_{2\to 1}(c_2)$ and $c_3 = \mathsf{map}_{2\to 3}(c_2)$.

$$\mathtt{Agg}(c_1, c_2, c_3) = \mathtt{Agg}(c_1, c_2, c_3)$$

- *Axiom 2.* Start with the axiom,

$$\mathsf{proj}_2(\mathtt{agg})\ \mathsf{map}_2^1\ \mathsf{proj}_1(\mathtt{agg}) \implies \mathsf{proj}_2(\mathtt{op}(\mathtt{agg}))\ \mathsf{map}_2^1\ \mathsf{proj}_1(\mathtt{op}(\mathtt{agg}))$$

and prove the consequent,

$$\mathsf{proj}_2(\mathtt{op}(\mathtt{agg}))\ \mathsf{map}_2^1\ \mathsf{proj}_1(\mathtt{op}(\mathtt{agg}))$$

by first unfolding the definition of op.

$$\mathsf{proj}_2(\mathsf{proj}_2^{-1}(\mathtt{op}_2(\mathsf{proj}_2(\mathtt{agg}))))\ \mathsf{map}_2^1\ \mathsf{proj}_1(\mathsf{proj}_2^{-1}(\mathtt{op}_2(\mathsf{proj}_2(\mathtt{agg}))))$$

Then simplify, using: $\mathsf{proj}_2(\mathsf{proj}_2^{-1}(x)) == x$ and $\mathsf{proj}_1(\mathsf{proj}_2^{-1}(x)) == \mathsf{map}_{2\to 1}(x)$.

$$\mathtt{op}_2(\mathsf{proj}_2(\mathtt{agg}))\ \mathsf{map}_2^1\ \mathsf{map}_{2\to 1}(\mathtt{op}_2(\mathsf{proj}_2(\mathtt{agg})))$$

Factor out the common subexpression to make it more obvious.

$$x\ \mathsf{map}_2^1\ \mathsf{map}_{2\to 1}(x) \text{ where } x = \mathtt{op}_2(\mathsf{proj}_2(\mathtt{agg}))$$

This is true by definition of map_2^1.

- *Axiom 3.* Showing that the third axiom holds is similar to the proof for Axiom 2.
- *Axiom 4.*
  Start with the axiom,

$$\mathsf{proj}_3(\mathtt{agg})\ \mathsf{map}_3^1\ \mathsf{proj}_1(\mathtt{agg}) \implies \mathsf{proj}_3(\mathtt{op}(\mathtt{agg}))\ \mathsf{map}_3^1\ \mathsf{proj}_1(\mathtt{op}(\mathtt{agg}))$$

and prove the consequent,

$$\mathsf{proj}_3(\mathtt{op}(\mathtt{agg}))\ \mathsf{map}_3^1\ \mathsf{proj}_1(\mathtt{op}(\mathtt{agg}))$$

by first unfolding op.

$$proj_3(proj_2^{-1}(op_2(proj_2(agg)))) \ map_3^1 \ proj_1(proj_2^{-1}(op_2(proj_2(agg))))$$

Then fold the definitions of $map_{2\to3}$ and $map_{2\to1}$,

$$map_{2\to3}(op_2(proj_2(agg))) \ map_3^1 \ map_{2\to1}(op_2(proj_2(agg)))$$

and factor out the common subexpression.

$$map_{2\to3}(x) \ map_3^1 \ map_{2\to1}(x) \text{ where } x = op_2(proj_2(agg))$$

Substitute $map_{3\to1}(map_{2\to3}(x))$ for $map_{2\to1}(x)$,

$$map_{2\to3}(x) \ map_3^1 \ map_{3\to1}(map_{2\to3}(x)) \text{ where } x = op_2(proj_2(agg))$$

and factor out the common subexpression.

$$y \ map_3^1 \ map_{3\to1}(y) \text{ where } x = op_2(proj_2(agg)) \text{ and } y = map_{2\to3}(x)$$

This is true by definition of map_3^1.

Reimplementing the Operations

Given a definition of the operation in one component, we derive alternative implementations for the other components using data transform definitions and then define the aggregate operation in terms of these component operations.

Given: op_2, $map_{2\to1}$, $map_{3\to2}$
local
$op_1(map_{2\to1}(c_2)) \Leftarrow map_{2\to1}(op_2(c_2))$
$map_{3\to2}(op_3(c_3)) \Leftarrow op_2(map_{3\to2}(c_3))$
in
$op(Agg(c_1, c_2, c_3)) \Leftarrow Agg(c_1', c_2', c_3')$
where $c_1' = op_1(c_1)$
and $c_2' = op_2(c_2)$
and $c_3' = op_3(c_3)$
end

- *Axiom 1.* Start with the axiom,

$$proj_2(op(Agg(c_1, c_2, c_3))) = op_2(proj_2(Agg(c_1, c_2, c_3))$$

and take the second projection on the righthand side.

$$\mathsf{proj}_2(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3))) \quad = \quad \mathsf{op}_2(c_2)$$

Unfold op and take the second projection on the lefthand side.

$$\mathsf{op}_2(c_2) \quad = \quad \mathsf{op}_2(c_2)$$

- *Axiom 2.* Start with the axiom:

$$\mathsf{proj}_2(\mathsf{Agg}(c_1, c_2, c_3))\ \mathsf{map}_2^1\ \mathsf{proj}_1(\mathsf{Agg}(c_1, c_2, c_3)) \implies \mathsf{proj}_2(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3)))\ \mathsf{map}_2^1\ \mathsf{proj}_1(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3)))$$

To prove the consequent,

$$\mathsf{proj}_2(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3)))\ \mathsf{map}_2^1\ \mathsf{proj}_1(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3)))$$

define an implementation of the relation as a compatibility map that maps one component into the other; that is: $x\ \mathsf{map}_2^1\ y \equiv y = \mathsf{map}_{2\to 1}(x)$.

$$\mathsf{proj}_1(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3))) \quad \stackrel{?}{=} \quad \mathsf{map}_{2\to 1}(\mathsf{proj}_2(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3))))$$

Unfold op and take the projection.

$$\mathsf{op}_1(c_1) \quad \stackrel{?}{=} \quad \mathsf{map}_{2\to 1}(\mathsf{op}_2(c_2))$$

Since the righthand side matches the body of op_1, fold the (expression procedure) definition of op_1,

$$\mathsf{op}_1(c_1) \quad \stackrel{?}{=} \quad \mathsf{op}_1(\mathsf{map}_{2\to 1}(c_2))$$

and then substitute c_1 for $\mathsf{map}_{2\to 1}(c_2)$ which is given by the antecedent.

$$\mathsf{op}_1(c_1) \quad = \quad \mathsf{op}_1(c_1)$$

- *Axiom 3.* Start with the axiom:

$$\mathsf{proj}_3(\mathsf{Agg}(c_1, c_2, c_3))\ \mathsf{map}_3^2\ \mathsf{proj}_2(\mathsf{Agg}(c_1, c_2, c_3)) \implies \mathsf{proj}_3(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3)))\ \mathsf{map}_3^2\ \mathsf{proj}_2(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3)))$$

As above, to prove the consequent,

$$\mathsf{proj}_3(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3)))\ \mathsf{map}_3^2\ \mathsf{proj}_2(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3)))$$

define the relation in terms of a compatibility map; that is:
$x\ \mathsf{map}_3^2\ y \equiv y = \mathsf{map}_{3\to 2}(x)$.

$$\mathsf{proj}_2(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3))) \stackrel{?}{=} \mathsf{map}_{3\to 2}(\mathsf{proj}_3(\mathsf{op}(\mathsf{Agg}(c_1, c_2, c_3))))$$

Unfold op and take the projection.

$$\mathsf{op}_2(c_2) \stackrel{?}{=} \mathsf{map}_{3\to 2}(\mathsf{op}_3(c_3))$$

Since the righthand side matches the definition of op_3, unfold the (expression procedure) definition of op_3,

$$\mathsf{op}_2(c_2) \stackrel{?}{=} \mathsf{op}_2(\mathsf{map}_{3\to 2}(c_3))$$

and then substitute c_2 for $\mathsf{map}_{3\to 2}(c_3)$ which is given by the antecedent.

$$\mathsf{op}_2(c_2) = \mathsf{op}_2(c_2)$$

- *Axiom 4.*

Start with the axiom:

$$\mathsf{proj}_3(\mathsf{agg})\ \mathsf{map}_3^1\ \mathsf{proj}_1(\mathsf{agg}) \implies \mathsf{proj}_3(\mathsf{op}(\mathsf{agg}))\ \mathsf{map}_3^1\ \mathsf{proj}_1(\mathsf{op}(\mathsf{agg}))$$

To prove the consequent,

$$\mathsf{proj}_3(\mathsf{op}(\mathsf{agg}))\ \mathsf{map}_3^1\ \mathsf{proj}_1(\mathsf{op}(\mathsf{agg}))$$

first define the relation in terms of a compatibility map:
$x\ \mathsf{map}_3^1\ y \equiv y = \mathsf{map}_{2\to 1}(\mathsf{map}_{3\to 2}(x))$.

$$\mathsf{proj}_1(\mathsf{op}(\mathsf{agg})) \stackrel{?}{=} \mathsf{map}_{2\to 1}(\mathsf{map}_{3\to 2}(\mathsf{proj}_3(\mathsf{op}(\mathsf{agg}))))$$

Unfold op and take the projection.

$$\mathsf{op}_1(c_1) \stackrel{?}{=} \mathsf{map}_{2\to 1}(\mathsf{map}_{3\to 2}(\mathsf{op}_3(c_3)))$$

Then unfold the (expression procedure) definition of op_3,

$$\mathsf{op}_1(c_1) \stackrel{?}{=} \mathsf{map}_{2\to 1}(\mathsf{op}_2(\mathsf{map}_{3\to 2}(c_3)))$$

and fold the (expression procedure) definition of op_1.

$$\mathsf{op}_1(c_1) \stackrel{?}{=} \mathsf{op}_1(\mathsf{map}_{2\to 1}(\mathsf{map}_{3\to 2}(c_3)))$$

Simplify, using: $c_1 = \mathsf{map}_{2\to 1}(\mathsf{map}_{3\to 2}(c_3))$.

$$\mathsf{op}_1(c_1) = \mathsf{op}_1(c_1)$$

Showing Transitivity

Here we relax the restriction that requires any two components to be directly connected by a compatibility map, to simply requiring a connection, possibly through some number of intermediate components. In all of the cases, the proof for Axiom 1 in this section is identical to the proof for Axiom 1 in the preceding section. We focus on Axiom 3 since it is the interesting case where the consistency relation cannot be defined in terms of a single compatibility map, and so an intermediate component must be used. The proofs for the other axioms are similar to the ones in the preceding section since there are direct translation functions in these cases.

Case 1.

Given: op_2, $map_{2\to1}$, $map_{1\to3}$

local

$$op_1(map_{2\to1}(c_2)) \Leftarrow map_{2\to1}(op_2(c_2))$$
$$op_3(map_{1\to3}(c_3)) \Leftarrow map_{1\to3}(op_1(c_3))$$

in

$$op(Agg(c_1, c_2, c_3)) \Leftarrow Agg(c_1', c_2', c_3')$$
$$\textbf{where } c_1' = op_1(c_1)$$
$$\textbf{and } c_2' = op_2(c_2)$$
$$\textbf{and } c_3' = op_3(c_3)$$

end

Unlike the proof for Axiom 3 in the preceding section, to prove the consequent,

$$proj_3(op(Agg(c_1, c_2, c_3))) \; map_3^2 \; proj_2(op(Agg(c_1, c_2, c_3)))$$

we define the relation in terms of the composition of the compatibility maps. The components are "related" if c_2 can be translated into c_3 using the composition of the compatibility maps.

$$proj_3(op(Agg(c_1, c_2, c_3))) \stackrel{?}{=} map_{1\to3}(map_{2\to1}(proj_2(op(Agg(c_1, c_2, c_3)))))$$

Unfold op and take the projection.

$$op_3(c_3) \stackrel{?}{=} map_{1\to3}(map_{2\to1}(op_2(c_2)))$$

Fold the definition of op_1,

$$op_3(c_3) \stackrel{?}{=} map_{1\to3}(op_1(map_{2\to1}(c_2)))$$

and then fold the definition of op_3.

$$op_3(c_3) \stackrel{?}{=} op_3(map_{1\to3}(map_{2\to1}(c_2)))$$

Simplify using: $c_3 = map_{1\to3}(map_{2\to1}(c_2))$ from the antecedent.

$$op_3(c_3) = op_3(c_3)$$

Case 2. Similar to Case 1.

Case 3.

Given: op_2, $map_{2\to 1}$, $map_{3\to 1}$

local

$$op_1(map_{2\to 1}(c_2)) \Leftarrow map_{2\to 1}(op_2(c_2))$$
$$map_{3\to 1}(op_3(c_3)) \Leftarrow op_1(map_{3\to 1}(c_3))$$

in

$$op(Agg(c_1, c_2, c_3)) \Leftarrow Agg(c_1', c_2', c_3')$$
$$\textbf{where } c_1' = op_1(c_1)$$
$$\textbf{and } c_2' = op_2(c_2)$$
$$\textbf{and } c_3' = op_3(c_3)$$

end

To prove that this definition satisfies the axioms, we need only reconsider the proof for Axiom 3 (since the others remain the same). Unlike the proof for Axiom 3 in the preceding section, to prove the consequent,

$$proj_3(op(Agg(c_1, c_2, c_3))) \; map_3^2 \; proj_2(op(Agg(c_1, c_2, c_3)))$$

we define the relation in terms of a pair of compatibility maps. The components are "related" if they both can be translated into some common form.

$$map_{2\to 1}(proj_2(op(Agg(c_1, c_2, c_3)))) \stackrel{?}{=} map_{3\to 1}(proj_3(op(Agg(c_1, c_2, c_3))))$$

Unfold op and take the projection.

$$map_{2\to 1}(op_2(c_2)) \stackrel{?}{=} map_{3\to 1}(op_3(c_3))$$

Fold the definition of op_1 on the lefthand side, and unfold the definition of op_3 on the righthand side.

$$op_1(map_{2\to 1}(c_2)) \stackrel{?}{=} op_1(map_{3\to 1}(c_3))$$

Simplify, using: $map_{2\to 1}(c_2) = map_{3\to 1}(c_3)$ from the antecedent.

$$op_1(map_{3\to 1}(c_3)) = op_1(map_{3\to 1}(c_3))$$

Case 4.

Given: op_2, $map_{1\to 2}$, $map_{1\to 3}$

local

$map_{1\to 2}(op_1(c_1)) \Leftarrow op_2(map_{1\to 2}(c_1))$

$op_3(map_{1\to 3}(c_1)) \Leftarrow map_{1\to 3}(op_1(c_1))$

in

$op(Agg(c_1, c_2, c_3)) \Leftarrow Agg(c_1', c_2', c_3')$

where $c_1' = op_1(c_1)$

and $c_2' = op_2(c_2)$

and $c_3' = op_3(c_3)$

end

As for case 3, we need only consider the proof for Axiom 3.

$$proj_3(op(Agg(c_1, c_2, c_3))) \; map_3^2 \; proj_2(op(Agg(c_1, c_2, c_3)))$$

We define the relation in terms of some intermediate,

$$\exists x \,.\, map_{1\to 2}(x) = proj_2(op(Agg(c_1, c_2, c_3))) \text{ AND } map_{1\to 3}(x) = proj_3(op(Agg(c_1, c_2, c_3)))$$

and take the appropriate projections.

$$\exists x \,.\, map_{1\to 2}(x) = op_2(c_2) \text{ AND } map_{1\to 3}(x) = op_3(c_3)$$

The components are consistent by definition, that is: $map_{1\to 2}(c_1) = c_2$ and $map_{1\to 3}(c_1) = c_3$. The antecedent states that the c_1 component is indeed the same in each equation.

$$\exists x \,.\, map_{1\to 2}(x) = op_2(map_{1\to 2}(c_1)) \text{ AND } map_{1\to 3}(x) = op_3(map_{1\to 3}(c_1))$$

Fold the body of op_1 and unfold the definition of op_3.

$$\exists x \,.\, map_{1\to 2}(x) = map_{1\to 2}(op_1(c_1)) \text{ AND } map_{1\to 3}(x) = map_{1\to 3}(op_1(c_1))$$

This is true when x is equal to $op_1(c_1)$.

Incrementally building the aggregate.

This section demonstrates how the incremental definition shown below satisfies the definition presented in Figure 6.8. This approach is a verification presentation where the new definition is "invented" and then the equivalence between the old and new definitions is shown using the usual symbol manipulations of fold and unfold.

Given: op_2, $map_{2\to 1}$, $map_{3\to 1}$

local

$span(c_2) \Leftarrow Agg_i(c_1, c_2)$ **where** $c_1 = map_{2\to 1}(c_2)$

$unspan(Agg(c_1, c_2, c_3)) \Leftarrow Agg_i(\{ c_1 \mid map_{3\to 1}(c_3) \}, c_2)$

in

$op_i(span(c_2)) \Leftarrow span(op_2(c_2))$

$unspan(op(Agg(c_1, c_2, c_3))) \Leftarrow op_i(unspan(Agg(c_1, c_2, c_3)))$

end

The following insight is useful in coming up with the aggregate definition. In the previous definition of op, we saw that it contained the following subexpression.

$$\ldots c_1' = op_1(c_1) \text{ and } c_2' = op_2(c_2) \ldots$$

We can transform a pair of functions, each returning a single value, into a single function returning a pair of values. That is, transform $c_1', c_2' = op_1(c_1), op_2(c_2)$ into $c_1', c_2' = op_i(c_1, c_2)$. Notice that the components are now computed together (in a single function), so that, for example, c_2 is available to compute the new value of c_1, which the designer might use if it increases efficiency. We can then repeat the process and combine the result with the third component c_3.

Here we start with the new definition and follow a sequence of transformations that demonstrate how it is equivalent to the old one (in Figure 6.8).

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow op_i(\text{unspan}(\text{Agg}(c_1, c_2, c_3)))$$

Unfold unspan on the righthand side.

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow op_i(\text{Agg}_i(\{ c_1 \mid map_{3\to 1}(c_3) \}, c_2))$$

The components are consistent; that is: $c_1 = map_{3\to 1}(c_3)$ and $c_1 = map_{2\to 1}(c_2)$.

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow op_i(\text{Agg}_i(\{ c_1 \mid map_{2\to 1}(c_2) \}, c_2))$$

Fold span on the righthand side,

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow op_i(\text{span}(c_2))$$

unfold op_i,

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow \text{Agg}_i(map_{2\to 1}(op_2(c_2)), op_2(c_2))$$

and then fold the definition of op_1.

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow \text{Agg}_i(op_1(map_{2\to 1}(c_2)), op_2(c_2))$$

The components are consistent; that is: $c_1 = map_{3\to 1}(c_3)$ and $c_1 = map_{2\to 1}(c_2)$.

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow \text{Agg}_i(op_1(map_{3\to 1}(c_3)), op_2(c_2))$$

Introduce the equality, $\{ x \mid x \}$. If x is valid, then both alternatives are valid ways to compute the value.

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow \text{Agg}_i(\{ op_1(map_{3\to 1}(c_3)) \mid op_1(map_{3\to 1}(c_3)) \}, op_2(c_2))$$

The components are consistent; that is: $c_1 = map_{3\to 1}(c_3)$.

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow \text{Agg}_i(\{ op_1(c_1) \mid op_1(map_{3\to 1}(c_3)) \}, op_2(c_2))$$

Fold the definition of op_3,

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow \text{Agg}_i(\{ op_1(c_1) \mid map_{3\to 1}(op_3(c_3)) \}, op_2(c_2))$$

fold unspan,

$$\text{unspan}(\text{op}(\text{Agg}(c_1, c_2, c_3))) \Leftarrow \text{unspan}(\text{Agg}(op_1(c_1), op_2(c_2), op_3(c_3)))$$

and then choose a solution.

$$\text{op}(\text{Agg}(c_1, c_2, c_3)) \Leftarrow \text{Agg}(op_1(c_1), op_2(c_2), op_3(c_3))$$